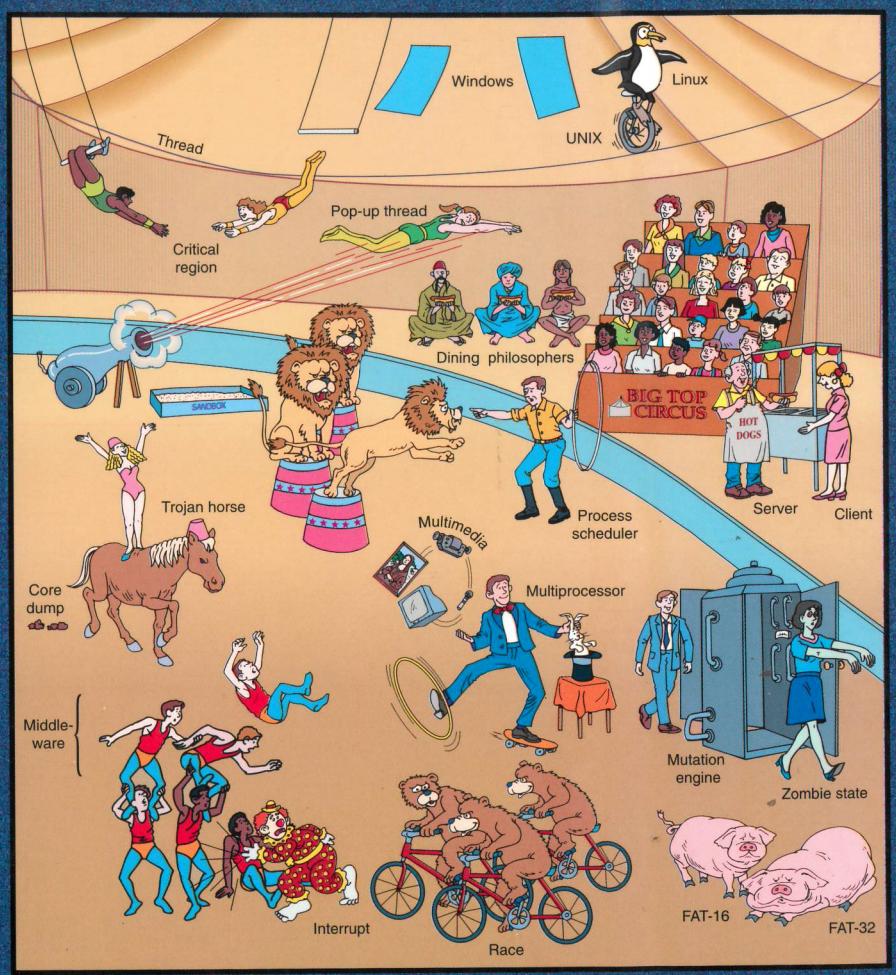


MODERN OPERATING SYSTEMS





SECOND EDITION



Andrew S. Tanenbaum

Spotify USA Inc. Exhibit 1009 Page 1

MODERN
OPERATING SYSTEMS

SECOND EDITION

Spotify USA Inc. Exhibit 1009 Page 2

Other bestselling titles by Andrew S. Tanenbaum

Structured Computer Organization, 4th edition

This widely-read classic, now in its fourth edition, provides the ideal introduction to computer architecture. It covers the topic in an easy-to-understand way, bottom up. There is a chapter on digital logic for beginners, followed by chapters on microarchitecture, the instruction set architecture level, operating systems, assembly language, and parallel computer architectures.

Computer Networks, 3rd edition

This widely-read classic, now in its third edition, provides the ideal introduction to today's and tomorrow's networks. It explains in detail how modern networks are structured. Starting with the physical layer and working up to the application layer, the book covers a vast number of important topics, including wireless communication, fiber optics, data link protocols, Ethernet, routing algorithms, network performance, security, DNS, electronic mail, USENET news, the World Wide Web, and multimedia. The book has especially thorough coverage of TCP/IP and the Internet.

Operating Systems: Design and Implementation, 2nd edition

This popular text on operating systems is the only book covering both the principles of operating systems and their application to a real system. All the traditional operating systems topics are covered in detail. In addition, the principles are carefully illustrated with MINIX, a free POSIX-based UNIX-like operating system for personal computers. Each book contains a free CD-ROM containing the complete MINIX system, including all the source code. The source code is listed in an appendix to the book amd explained in detail in the text.

Distributed Operating Systems

This text covers the fundamental concepts of distributed operating systems. Key topics include communication and synchronization, processes and processors, distributed shared memory, distributed file systems, and distributed real-time systems. The principles are illustrated using four chapter-long examples.

Spotify USA Inc. Exhibit 1009 Page 3

MODERN OPERATING SYSTEMS

SECOND EDITION

ANDREW S. TANENBAUM

Vrije Universiteit
Amsterdam, The Netherlands



PRENTICE HALL
UPPER SADDLE RIVER, NEW JERSEY 07458

Spotify USA Inc. Exhibit 1009 Page 4

Library of Congress Cataloging-in-Publication Data
Tanenbaum, Andrew S.
Modern operating systems / Andrew S. Tanenbaum.--2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-031358-0
1.Operating systems (Computers) I. Title

QA76.76.O63 T359 2001
005.4'3--dc21

Vice President and Editorial Director, ECS: ***Marcia Horton***
Publisher: ***Alan Apt***
Associate Editor: ***Toni D. Holm***
Editorial Assistant: ***Amy K. Todd***
Vice President and Director of Production and Manufacturing, ESM: ***David W. Riccardi***
Executive Managing Editor: ***Vince O'Brien***
Managing Editor: ***David A. George***
Senior Production Editor: ***Camille Trentacoste***
Director of Creative Services: ***Paul Belfanti***
Creative Director: ***Carole Anson***
Art Director: ***Heather Scott***
Cover Art: ***Don Martinetti***
Cover Design: ***Joseph Sengotta***
Assistant to Art Director: ***John Christiana***
Interior Illustration: ***Patricia Gutièrrez***
Interior Design and Typesetting; Cover Illustration Concept: ***Andrew S. Tanenbaum***
Manufacturing Manager: ***Trudy Pisciotti***
Manufacturing Buyer: ***Pat Brown***
Marketing Manager: ***Jennie Burger***



© 2001 by Prentice-Hall, Inc.
Upper Saddle River, New Jersey 07458

The authors and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in this book. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks and registered trademarks. Where those designations appear in this book, and Prentice Hall and the authors were aware of a trademark claim, the designations have been printed in initial caps or all caps. All product names mentioned remain trademarks or registered trademarks of their respective owners.

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5

ISBN 0-13-031358-0

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Pearson Education Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Spotify USA Inc. Exhibit 1009 Page 5

CONTENTS

PREFACE xvi

1 INTRODUCTION 1

1.1. WHAT IS AN OPERATING SYSTEM? 3
- 1.1.1. The Operating System as an Extended Machine 3
- 1.1.2. The Operating System as a Resource Manager 5

1.2. HISTORY OF OPERATING SYSTEMS 6
- 1.2.1. The First Generation (1945-55) 6
- 1.2.2. The Second Generation (1955-65) 7
- 1.2.3. The Third Generation (1965-1980) 9
- 1.2.4. The Fourth Generation (1980-Present) 13
- 1.2.5. Ontogeny Recapitulates Phylogeny 16

1.3. THE OPERATING SYSTEM ZOO 18
- 1.3.1. Mainframe Operating Systems 18
- 1.3.2. Server Operating Systems 19
- 1.3.3. Multiprocessor Operating Systems 19
- 1.3.4. Personal Computer Operating Systems 19
- 1.3.5. Real-Time Operating Systems 19
- 1.3.6. Embedded Operating Systems 20
- 1.3.7. Smart Card Operating Systems 20

Spotify USA Inc. Exhibit 1009 Page 6

1.4. COMPUTER HARDWARE REVIEW 20
1.4.1. Processors 21
1.4.2. Memory 23
1.4.3. I/O Devices 28
1.4.4. Buses 31

1.5. OPERATING SYSTEM CONCEPTS 34
1.5.1. Processes 34
1.5.2. Deadlocks 36
1.5.3. Memory Management 37
1.5.4. Input/Output 38
1.5.5. Files 38
1.5.6. Security 41
1.5.7. The Shell 41
1.5.8. Recycling of Concepts 43

1.6. SYSTEM CALLS 44
1.6.1. System Calls for Process Management 48
1.6.2. System Calls for File Management 50
1.6.3. System Calls for Directory Management 51
1.6.4. Miscellaneous System Calls 53
1.6.5. The Windows Win32 API 53

1.7. OPERATING SYSTEM STRUCTURE 56
1.7.1. Monolithic Systems 56
1.7.2. Layered Systems 57
1.7.3. Virtual Machines 59
1.7.4. Exokernels 61
1.7.5. Client-Server Model 61

1.8. RESEARCH ON OPERATING SYSTEMS 63

1.9. OUTLINE OF THE REST OF THIS BOOK 65

1.10. METRIC UNITS 66

1.11. SUMMARY 67

2 PROCESSES AND THREADS 71

2.1. PROCESSES 71
- 2.1.1. The Process Model 72
- 2.1.2. Process Creation 73
- 2.1.3. Process Termination 75
- 2.1.4. Process Hierarchies 76
- 2.1.5. Process States 77
- 2.1.6. Implementation of Processes 79

2.2. THREADS 81
- 2.2.1. The Thread Model 81
- 2.2.2. Thread Usage 85
- 2.2.3. Implementing Threads in User Space 90
- 2.2.4. Implementing Threads in the Kernel 93
- 2.2.5. Hybrid Implementations 94
- 2.2.6. Scheduler Activations 94
- 2.2.7. Pop-Up Threads 96
- 2.2.8. Making Single-Threaded Code Multithreaded 97

2.3. INTERPROCESS COMMUNICATION 100
- 2.3.1. Race Conditions 100
- 2.3.2. Critical Regions 102
- 2.3.3. Mutual Exclusion with Busy Waiting 103
- 2.3.4. Sleep and Wakeup 108
- 2.3.5. Semaphores 110
- 2.3.6. Mutexes 113
- 2.3.7. Monitors 115
- 2.3.8. Message Passing 119
- 2.3.9. Barriers 123

2.4. CLASSICAL IPC PROBLEMS 124
- 2.4.1. The Dining Philosophers Problem 125
- 2.4.2. The Readers and Writers Problem 128
- 2.4.3. The Sleeping Barber Problem 129

2.5. SCHEDULING 132
- 2.5.1. Introduction to Scheduling 132
- 2.5.2. Scheduling in Batch Systems 138
- 2.5.3. Scheduling in Interactive Systems 142
- 2.5.4. Scheduling in Real-Time Systems 148
- 2.5.5. Policy versus Mechanism 149
- 2.5.6. Thread Scheduling 150

2.6. RESEARCH ON PROCESSES AND THREADS 151

2.7. SUMMARY 152

3 DEADLOCKS 159

3.1. RESOURCES 160
3.1.1. Preemptable and Nonpreemptable Resources 160
3.1.2. Resource Acquisition 161

3.2. INTRODUCTION TO DEADLOCKS 163
3.2.1. Conditions for Deadlock 164
3.2.2. Deadlock Modeling 164

3.3. THE OSTRICH ALGORITHM 167

3.4. DEADLOCK DETECTION AND RECOVERY 168
3.4.1. Deadlock Detection with One Resource of Each Type 168
3.4.2. Deadlock Detection with Multiple Resource of Each Type 171
3.4.3. Recovery from Deadlock 173

3.5. DEADLOCK AVOIDANCE 175
3.5.1. Resource Trajectories 175
3.5.2. Safe and Unsafe States 176
3.5.3. The Banker's Algorithm for a Single Resource 178
3.5.4. The Banker's Algorithm for Multiple Resources 179

3.6. DEADLOCK PREVENTION 180
3.6.1. Attacking the Mutual Exclusion Condition 180
3.6.2. Attacking the Hold and Wait Condition 181
3.6.3. Attacking the No Preemption Condition 182
3.6.4. Attacking the Circular Wait Condition 182

3.7. OTHER ISSUES 183
3.7.1. Two-Phase Locking 183
3.7.2. Nonresource Deadlocks 184
3.7.3. Starvation 184

3.8. RESEARCH ON DEADLOCKS 185

3.9. SUMMARY 185

Spotify USA Inc. Exhibit 1009 Page 9

4 MEMORY MANAGEMENT 189

4.1. BASIC MEMORY MANAGEMENT 190
- 4.1.1. Monoprogramming without Swapping or Paging 190
- 4.1.2. Multiprogramming with Fixed Partitions 191
- 4.1.3. Modeling Multiprogramming 192
- 4.1.4. Analysis of Multiprogramming System Performance 194
- 4.1.5. Relocation and Protection 194

4.2. SWAPPING 196
- 4.2.1. Memory Management with Bitmaps 199
- 4.2.2. Memory Management with Linked Lists 200

4.3. VIRTUAL MEMORY 202
- 4.3.1. Paging 202
- 4.3.2. Page Tables 205
- 4.3.3. TLBs—Translation Lookaside Buffers 211
- 4.3.4. Inverted Page Tables 213

4.4. PAGE REPLACEMENT ALGORITHMS 214
- 4.4.1. The Optimal Page Replacement Algorithm 215
- 4.4.2. The Not Recently Used Page Replacement Algorithm 216
- 4.4.3. The First-In, First-Out 217
- 4.4.4. The Second Chance Page Replacement Algorithm 217
- 4.4.5. The Clock Page Replacement Algorithm 218
- 4.4.6. The Least Recently Used 218
- 4.4.7. Simulating LRU in Software 220
- 4.4.8. The Working Set Page Replacement Algorithm 222
- 4.4.9. The WSClock Page Replacement Algorithm 225
- 4.4.:. Summary of Page Replacement Algorithms 227

4.5. MODELING PAGE REPLACEMENT ALGORITHMS 228
- 4.5.1. Belady's Anomaly 229
- 4.5.2. Stack Algorithms 229
- 4.5.3. The Distance String 232
- 4.5.4. Predicting Page Fault Rates 233

4.6. DESIGN ISSUES FOR PAGING SYSTEMS 234
- 4.6.1. Local versus Global Allocation Policies 234
- 4.6.2. Load Control 236
- 4.6.3. Page Size 237
- 4.6.4. Separate Instruction and Data Spaces 239

Spotify USA Inc. Exhibit 1009 Page 10

4.6.5. Shared Pages 239
4.6.6. Cleaning Policy 241
4.6.7. Virtual Memory Interface 241

4.7. IMPLEMENTATION ISSUES 242
4.7.1. Operating System Involvement with Paging 242
4.7.2. Page Fault Handling 243
4.7.3. Instruction Backup 244
4.7.4. Locking Pages in Memory 246
4.7.5. Backing Store 246
4.7.6. Separation of Policy and Mechanism 247

4.8. SEGMENTATION 249
4.8.1. Implementation of Pure Segmentation 253
4.8.2. Segmentation with Paging: MULTICS 254
4.8.3. Segmentation with Paging: The Intel Pentium 257

4.9. RESEARCH ON MEMORY MANAGEMENT 262

4.10. SUMMARY 262

5 INPUT/OUTPUT 269

5.1. PRINCIPLES OF I/O HARDWARE 269
5.1.1. I/O Devices 270
5.1.2. Device Controllers 271
5.1.3. Memory-Mapped I/O 272
5.1.4. Direct Memory Access 276
5.1.5. Interrupts Revisited 279

5.2. PRINCIPLES OF I/O SOFTWARE 282
5.2.1. Goals of the I/O Software 283
5.2.2. Programmed I/O 284
5.2.3. Interrupt-Driven I/O 286
5.2.4. I/O Using DMA 287

5.3. I/O SOFTWARE LAYERS 287
5.3.1. Interrupt Handlers 287
5.3.2. Device Drivers 289

Spotify USA Inc. Exhibit 1009 Page 11

5.3.3. Device-Independent I/O Software 292
5.3.4. User-Space I/O Software 298

5.4. DISKS 300
5.4.1. Disk Hardware 300
5.4.2. Disk Formatting 315
5.4.3. Disk Arm Scheduling Algorithms 318
5.4.4. Error Handling 322
5.4.5. Stable Storage 324

5.5. CLOCKS 327
5.5.1. Clock Hardware 328
5.5.2. Clock Software 329
5.5.3. Soft Timers 332

5.6. CHARACTER-ORIENTED TERMINALS 333
5.6.1. RS-232 Terminal Hardware 334
5.6.2. Input Software 336
5.6.3. Output Software 341

5.7. GRAPHICAL USER INTERFACES 342
5.7.1. Personal Computer Keyboard, Mouse, and Display Hardware 343
5.7.2. Input Software 347
5.7.3. Output Software for Windows 347

5.8. NETWORK TERMINALS 355
5.8.1. The X Window System 356
5.8.2. The SLIM Network Terminal 360

5.9. POWER MANAGEMENT 363
5.9.1. Hardware Issues 364
5.9.2. Operating System Issues 365
5.9.3. Degraded Operation 370

5.10. RESEARCH ON INPUT/OUTPUT 371

5.11. SUMMARY 372

6 FILE SYSTEMS 379

6.1. FILES 380
- 6.1.1. File Naming 380
- 6.1.2. File Structure 382
- 6.1.3. File Types 383
- 6.1.4. File Access 385
- 6.1.5. File Attributes 386
- 6.1.6. File Operations 387
- 6.1.7. An Example Program Using File System Calls 389
- 6.1.8. Memory-Mapped Files 391

6.2. DIRECTORIES 393
- 6.2.1. Single-Level Directory Systems 393
- 6.2.2. Two-level Directory Systems 394
- 6.2.3. Hierarchical Directory Systems 395
- 6.2.4. Path Names 395
- 6.2.5. Directory Operations 398

6.3. FILE SYSTEM IMPLEMENTATION 399
- 6.3.1. File System Layout 399
- 6.3.2. Implementing Files 400
- 6.3.3. Implementing Directories 405
- 6.3.4. Shared Files 408
- 6.3.5. Disk Space Management 410
- 6.3.6. File System Reliability 416
- 6.3.7. File System Performance 424
- 6.3.8. Log-Structured File Systems 428

6.4. EXAMPLE FILE SYSTEMS 430
- 6.4.1. CD-ROM File Systems 430
- 6.4.2. The CP/M File System 435
- 6.4.3. The MS-DOS File System 438
- 6.4.4. The Windows 98 File System 442
- 6.4.5. The UNIX V7 File System 445

6.5. RESEARCH ON FILE SYSTEMS 448

6.6. SUMMARY 448

Spotify USA Inc. Exhibit 1009 Page 13

7 MULTIMEDIA OPERATING SYSTEMS 453

7.1. INTRODUCTION TO MULTIMEDIA 454

7.2. MULTIMEDIA FILES 458
7.2.1. Audio Encoding 459
7.2.2. Video Encoding 461

7.3. VIDEO COMPRESSION 463
7.3.1. The JPEG Standard 464
7.3.2. The MPEG Standard 467

7.4. MULTIMEDIA PROCESS SCHEDULING 469
7.4.1. Scheduling Homogeneous Processes 469
7.4.2. General Real-Time Scheduling 470
7.4.3. Rate Monotonic Scheduling 472
7.4.4. Earliest Deadline First Scheduling 473

7.5. MULTIMEDIA FILE SYSTEM PARADIGMS 475
7.5.1. VCR Control Functions 476
7.5.2. Near Video on Demand 478
7.5.3. Near Video on Demand with VCR Functions 479

7.6. FILE PLACEMENT 481
7.6.1. Placing a File on a Single Disk 481
7.6.2. Two Alternative File Organization Strategies 482
7.6.3. Placing Files for Near Video on Demand 486
7.6.4. Placing Multiple Files on a Single Disk 487
7.6.5. Placing Files on Multiple Disks 490

7.7. CACHING 492
7.7.1. Block Caching 492
7.7.2. File Caching 494

7.8. DISK SCHEDULING FOR MULTIMEDIA 494
7.8.1. Static Disk Scheduling 495
7.8.2. Dynamic Disk Scheduling 496

7.9. RESEARCH ON MULTIMEDIA 498

7.10. SUMMARY 499

8 MULTIPLE PROCESSOR SYSTEMS 503

8.1. MULTIPROCESSORS 506
- 8.1.1. Multiprocessor Hardware 506
- 8.1.2. Multiprocessor Operating System Types 513
- 8.1.3. Multiprocessor Synchronization 516
- 8.1.4. Multiprocessor Scheduling 521

8.2. MULTICOMPUTERS 526
- 8.2.1. Multicomputer Hardware 527
- 8.2.2. Low-Level Communication Software 531
- 8.2.3. User-Level Communication Software 534
- 8.2.4. Remote Procedure Call 537
- 8.2.5. Distributed Shared Memory 540
- 8.2.6. Multicomputer Scheduling 544
- 8.2.7. Load Balancing 545

8.3. DISTRIBUTED SYSTEMS 549
- 8.3.1. Network Hardware 551
- 8.3.2. Network Services and Protocols 553
- 8.3.3. Document-Based Middleware 558
- 8.3.4. File System-Based Middleware 559
- 8.3.5. Shared Object-Based Middleware 565
- 8.3.6. Coordination-Based Middleware 572

8.4. RESEARCH ON MULTIPLE PROCESSOR SYSTEMS 577

8.5. SUMMARY 577

9 SECURITY 583

9.1. THE SECURITY ENVIRONMENT 584
- 9.1.1. Threats 584
- 9.1.2. Intruders 585
- 9.1.3. Accidental Data Loss 586

9.2. BASICS OF CRYPTOGRAPHY 587
- 9.2.1. Secret-Key Cryptography 588
- 9.2.2. Public-Key Cryptography 588

Spotify USA Inc. Exhibit 1009 Page 15

9.2.3. One-Way Functions 589
9.2.4. Digital Signatures 590

9.3. USER AUTHENTICATION 591
9.3.1. Authentication Using Passwords 592
9.3.2. Authentication Using a Physical Object 601
9.3.3. Authentication Using Biometrics 603
9.3.4. Countermeasures 606

9.4. ATTACKS FROM INSIDE THE SYSTEM 606
9.4.1. Trojan Horses 607
9.4.2. Login Spoofing 608
9.4.3. Logic Bombs 609
9.4.4. Trap Doors 610
9.4.5. Buffer Overflow 610
9.4.6. Generic Security Attacks 613
9.4.7. Famous Security Flaws 614
9.4.8. Design Principles for Security 616

9.5. ATTACKS FROM OUTSIDE THE SYSTEM 617
9.5.1. Virus Damage Scenarios 618
9.5.2. How Viruses Work 619
9.5.3. How Viruses Spread 626
9.5.4. Antivirus and Anti-Antivirus Techniques 628
9.5.5. The Internet Worm 635
9.5.6. Mobile Code 637
9.5.7. Java Security 642

9.6. PROTECTION MECHANISMS 645
9.6.1. Protection Domains 645
9.6.2. Access Control Lists 647
9.6.3. Capabilities 650

9.7. TRUSTED SYSTEMS 653
9.7.1. Trusted Computing Base 654
9.7.2. Formal Models of Secure Systems 655
9.7.3. Multilevel Security 657
9.7.4. Orange Book Security 659
9.7.5. Covert Channels 661

9.8. RESEARCH ON SECURITY 665

9.9. SUMMARY 666

10 CASE STUDY 1: UNIX AND LINUX 671

10.1. HISTORY OF UNIX 672
- 10.1.1. UNICS 672
- 10.1.2. PDP-11 UNIX 673
- 10.1.3. Portable UNIX 674
- 10.1.4. Berkeley UNIX 675
- 10.1.5. Standard UNIX 676
- 10.1.6. MINIX 677
- 10.1.7. Linux 678

10.2. OVERVIEW OF UNIX 681
- 10.2.1. UNIX Goals 681
- 10.2.2. Interfaces to UNIX 682
- 10.2.3. The UNIX Shell 683
- 10.2.4. UNIX Utility Programs 686
- 10.2.5. Kernel Structure 687

10.3. PROCESSES IN UNIX 690
- 10.3.1. Fundamental Concepts 690
- 10.3.2. Process Management System Calls in UNIX 692
- 10.3.3. Implementation of Processes in UNIX 699
- 10.3.4. Booting UNIX 708

10.4. MEMORY MANAGEMENT IN UNIX 710
- 10.4.1. Fundamental Concepts 711
- 10.4.2. Memory Management System Calls in UNIX 714
- 10.4.3. Implementation of Memory Management in UNIX 715

10.5. INPUT/OUTPUT IN UNIX 723
- 10.5.1. Fundamental Concepts 724
- 10.5.2. Input/Output System Calls in UNIX 726
- 10.5.3. Implementation of Input/Output in UNIX 727
- 10.5.4. Streams 730

10.6. THE UNIX FILE SYSTEM 732
- 10.6.1. Fundamental Concepts 732
- 10.6.2. File System Calls in UNIX 736
- 10.6.3. Implementation of the UNIX File System 740
- 10.6.4. NFS: The Network File System 747

Spotify USA Inc. Exhibit 1009 Page 17

10.7. SECURITY IN UNIX 753
10.7.1. Fundamental Concepts 753
10.7.2. Security System Calls in UNIX 755
10.7.3. Implementation of Security in UNIX 756

10.8. SUMMARY 757

11 CASE STUDY 2: WINDOWS 2000 763

11.1. HISTORY OF WINDOWS 2000 763
11.1.1. MS-DOS 763
11.1.2. Windows 95/98/Me 764
11.1.3. Windows NT 765
11.1.4. Windows 2000 767

11.2. PROGRAMMING WINDOWS 2000 771
11.2.1. The Win32 Application Programming Interface 772
11.2.2. The Registry 774

11.3. SYSTEM STRUCTURE 778
11.3.1. Operating System Structure 778
11.3.2. Implementation of Objects 787
11.3.3. Environment Subsystems 792

11.4. PROCESSES AND THREADS IN WINDOWS 2000 796
11.4.1. Fundamental Concepts 796
11.4.2. Job, Process, Thread and Fiber Management API Calls 799
11.4.3. Implementation of Processes and Threads 802
11.4.4. MS-DOS Emulation 809
11.4.5. Booting Windows 2000 820

11.5. MEMORY MANAGEMENT 811
11.5.1. Fundamental Concepts 812
11.5.2. Memory Management System Calls 816
11.5.3. Implementation of Memory Management 817

11.6. INPUT/OUTPUT IN WINDOWS 2000 824
11.6.1. Fundamental Concepts 824
11.6.2. Input/Output API Calls 825
11.6.3. Implementation of I/O 827
11.6.4. Device Drivers 827

11.7. THE WINDOWS 2000 FILE SYSTEM 830
11.7.1. Fundamental Concepts 830
11.7.2. File System API Calls in Windows 2000 831
11.7.3. Implementation of the Windows 2000 File System 833

11.8. SECURITY IN WINDOWS 2000 844
11.8.1. Fundamental Concepts 845
11.8.2. Security API Calls 847
11.8.3. Implementation of Security 848

11.9. CACHING IN WINDOWS 2000 849

11.10. SUMMARY 851

12 OPERATING SYSTEM DESIGN 855

12.1. THE NATURE OF THE DESIGN PROBLEM 856
12.1.1. Goals 856
12.1.2. Why is it Hard to Design an Operating Systems? 857

12.2. INTERFACE DESIGN 859
12.2.1. Guiding Principles 859
12.2.2. Paradigms 861
12.2.3. The System Call Interface 864

12.3 IMPLEMENTATION 867
12.3.1. System Structure 867
12.3.2. Mechanism versus Policy 870
12.3.3. Orthogonality 871
12.3.4. Naming 872
12.3.5. Binding Time 874
12.3.6. Static versus Dynamic Structures 875
12.3.7. Top-Down versus Bottom-Up Implementation 876
12.3.8. Useful Techniques 877

12.4. PERFORMANCE 882
12.4.1. Why are Operating Systems Slow? 882
12.4.2. What Should be Optimized? 883
12.4.3. Space-Time Trade-offs 884
12.4.4. Caching 887

Spotify USA Inc. Exhibit 1009 Page 19

12.4.5. Hints 888
12.4.6. Exploiting Locality 888
12.4.7. Optimize the Common Case 889

12.5. PROJECT MANAGEMENT 889
12.5.1. The Mythical Man Month 890
12.5.2. Team Structure 891
12.5.3. The Role of Experience 893
12.5.4. No Silver Bullet 894

12.6. TRENDS IN OPERATING SYSTEM DESIGN 894
12.6.1. Large Address Space Operating Systems 894
12.6.2. Networking 895
12.6.3. Parallel and Distributed Systems 896
12.6.4. Multimedia 896
12.6.5. Battery-Powered Computers 896
12.6.6. Embedded Systems 897

12.7. SUMMARY 897

13 READING LIST AND BIBLIOGRAPHY 901

13.1. SUGGESTIONS FOR FURTHER READING 901
13.1.1. Introduction and General Works 902
13.1.2. Processes and Threads 902
13.1.3. Deadlocks 903
13.1.4. Memory Management 903
13.1.5. Input/Output 903
13.1.6. File Systems 904
13.1.7. Multimedia Operating Systems 905
13.1.8. Multiple Processor Systems 906
13.1.9. Security 907
13.1.10. UNIX and Linux 908
13.1.11. Windows 2000 909
13.1.12. Design Principles 910

13.2 ALPHABETICAL BIBLIOGRAPHY 911

INDEX 935

1

INTRODUCTION

A modern computer system consists of one or more processors, some main memory, disks, printers, a keyboard, a display, network interfaces, and other input/output devices. All in all, a complex system. Writing programs that keep track of all these components and use them correctly, let alone optimally, is an extremely difficult job. For this reason, computers are equipped with a layer of software called the **operating system**, whose job is to manage all these devices and provide user programs with a simpler interface to the hardware. These systems are the subject of this book.

The placement of the operating system is shown in Fig. 1-1. At the bottom is the hardware, which, in many cases, is itself composed of two or more levels (or layers). The lowest level contains physical devices, consisting of integrated circuit chips, wires, power supplies, cathode ray tubes, and similar physical devices. How these are constructed and how they work are the provinces of the electrical engineer.

Next comes the **microarchitecture level**, in which the physical devices are grouped together to form functional units. Typically this level contains some registers internal to the CPU (Central Processing Unit) and a data path containing an arithmetic logic unit. In each clock cycle, one or two operands are fetched from the registers and combined in the arithmetic logic unit (for example, by addition or Boolean AND). The result is stored in one or more registers. On some machines, the operation of the data path is controlled by software, called the **microprogram**. On other machines, it is controlled directly by hardware circuits.

Spotify USA Inc. Exhibit 1009 Page 21

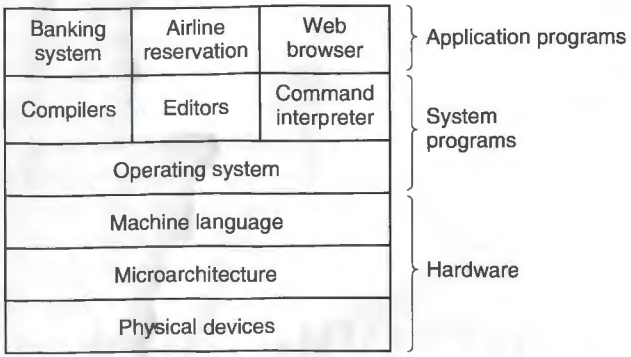


Figure 1-1. A computer system consists of hardware, system programs, and application programs.

The purpose of the data path is to execute some set of instructions. Some of these can be carried out in one data path cycle; others may require multiple data path cycles. These instructions may use registers or other hardware facilities. Together, the hardware and instructions visible to an assembly language programmer form the **ISA (Instruction Set Architecture)** level. This level is often called **machine language**.

The machine language typically has between 50 and 300 instructions, mostly for moving data around the machine, doing arithmetic, and comparing values. In this level, the input/output devices are controlled by loading values into special **device registers**. For example, a disk can be commanded to read by loading the values of the disk address, main memory address, byte count, and direction (read or write) into its registers. In practice, many more parameters are needed, and the status returned by the drive after an operation is highly complex. Furthermore, for many I/O (Input/Output) devices, timing plays an important role in the programming.

To hide this complexity, an operating system is provided. It consists of a layer of software that (partially) hides the hardware and gives the programmer a more convenient set of instructions to work with. For example, read block from file is conceptually simpler than having to worry about the details of moving disk heads, waiting for them to settle down, and so on.

On top of the operating system is the rest of the system software. Here we find the command interpreter (shell), window systems, compilers, editors, and similar application-independent programs. It is important to realize that these programs are definitely not part of the operating system, even though they are typically supplied by the computer manufacturer. This is a crucial, but subtle, point. The operating system is (usually) that portion of the software that runs in **kernel mode** or **supervisor mode**. It is protected from user tampering by the hardware (ignoring for the moment some older or low-end microprocessors that do not have

Spotify USA Inc. Exhibit 1009 Page 22

hardware protection at all). Compilers and editors run in **user mode**. If a user does not like a particular compiler, he† is free to write his own if he so chooses; he is not free to write his own clock interrupt handler, which is part of the operating system and is normally protected by hardware against attempts by users to modify it.

This distinction, however, is sometimes blurred in embedded systems (which may not have kernel mode) or interpreted systems (such as Java-based operating systems that use interpretation, not hardware, to separate the components). Still, for traditional computers, the operating system is what runs in kernel mode.

That said, in many systems there are programs that run in user mode but which help the operating system or perform privileged functions. For example, there is often a program that allows users to change their passwords. This program is not part of the operating system and does not run in kernel mode, but it clearly carries out a sensitive function and has to be protected in a special way.

In some systems, this idea is carried to an extreme form, and pieces of what is traditionally considered to be the operating system (such as the file system) run in user space. In such systems, it is difficult to draw a clear boundary. Everything running in kernel mode is clearly part of the operating system, but some programs running outside it are arguably also part of it, or at least closely associated with it.

Finally, above the system programs come the application programs. These programs are purchased or written by the users to solve their particular problems, such as word processing, spreadsheets, engineering calculations, or storing information in a database.

1.1 WHAT IS AN OPERATING SYSTEM?

Most computer users have had some experience with an operating system, but it is difficult to pin down precisely what an operating system is. Part of the problem is that operating systems perform two basically unrelated functions, extending the machine and managing resources, and depending on who is doing the talking, you hear mostly about one function or the other. Let us now look at both.

1.1.1 The Operating System as an Extended Machine

As mentioned earlier, the **architecture** (instruction set, memory organization, I/O, and bus structure) of most computers at the machine language level is primitive and awkward to program, especially for input/output. To make this point more concrete, let us briefly look at how floppy disk I/O is done using the NEC

† "He" should be read as "he or she" throughout the book.

PD765 compatible controller chips used on most Intel-based personal computers. (Throughout this book we will use the terms "floppy disk" and "diskette" interchangeably.) The PD765 has 16 commands, each specified by loading between 1 and 9 bytes into a device register. These commands are for reading and writing data, moving the disk arm, and formatting tracks, as well as initializing, sensing, resetting, and recalibrating the controller and the drives.

The most basic commands are read and write, each of which requires 13 parameters, packed into 9 bytes. These parameters specify such items as the address of the disk block to be read, the number of sectors per track, the recording mode used on the physical medium, the intersector gap spacing, and what to do with a deleted-data-address-mark. If you do not understand this mumbo jumbo, do not worry; that is precisely the point—it is rather esoteric. When the operation is completed, the controller chip returns 23 status and error fields packed into 7 bytes. As if this were not enough, the floppy disk programmer must also be constantly aware of whether the motor is on or off. If the motor is off, it must be turned on (with a long startup delay) before data can be read or written. The motor cannot be left on too long, however, or the floppy disk will wear out. The programmer is thus forced to deal with the trade-off between long startup delays versus wearing out floppy disks (and losing the data on them).

Without going into the *real* details, it should be clear that the average programmer probably does not want to get too intimately involved with the programming of floppy disks (or hard disks, which are just as complex and quite different). Instead, what the programmer wants is a simple, high-level abstraction to deal with. In the case of disks, a typical abstraction would be that the disk contains a collection of named files. Each file can be opened for reading or writing, then read or written, and finally closed. Details such as whether or not recording should use modified frequency modulation and what the current state of the motor is should not appear in the abstraction presented to the user.

The program that hides the truth about the hardware from the programmer and presents a nice, simple view of named files that can be read and written is, of course, the operating system. Just as the operating system shields the programmer from the disk hardware and presents a simple file-oriented interface, it also conceals a lot of unpleasant business concerning interrupts, timers, memory management, and other low-level features. In each case, the abstraction offered by the operating system is simpler and easier to use than that offered by the underlying hardware.

In this view, the function of the operating system is to present the user with the equivalent of an **extended machine** or **virtual machine** that is easier to program than the underlying hardware. How the operating system achieves this goal is a long story, which we will study in detail throughout this book. To summarize it in a nutshell, the operating system provides a variety of services that programs can obtain using special instructions called system calls. We will examine some of the more common system calls later in this chapter.

Spotify USA Inc. Exhibit 1009 Page 24

1.1.2 The Operating System as a Resource Manager

The concept of the operating system as primarily providing its users with a convenient interface is a top-down view. An alternative, bottom-up, view holds that the operating system is there to manage all the pieces of a complex system. Modern computers consist of processors, memories, timers, disks, mice, network interfaces, printers, and a wide variety of other devices. In the alternative view, the job of the operating system is to provide for an orderly and controlled allocation of the processors, memories, and I/O devices among the various programs competing for them.

Imagine what would happen if three programs running on some computer all tried to print their output simultaneously on the same printer. The first few lines of printout might be from program 1, the next few from program 2, then some from program 3, and so forth. The result would be chaos. The operating system can bring order to the potential chaos by buffering all the output destined for the printer on the disk. When one program is finished, the operating system can then copy its output from the disk file where it has been stored to the printer, while at the same time the other program can continue generating more output, oblivious to the fact that the output is not really going to the printer (yet).

When a computer (or network) has multiple users, the need for managing and protecting the memory, I/O devices, and other resources is even greater, since the users might otherwise interfere with one another. In addition, users often need to share not only hardware, but information (files, databases, etc.) as well. In short, this view of the operating system holds that its primary task is to keep track of who is using which resource, to grant resource requests, to account for usage, and to mediate conflicting requests from different programs and users.

Resource management includes multiplexing (sharing) resources in two ways: in time and in space. When a resource is time multiplexed, different programs or users take turns using it. First one of them gets to use the resource, then another, and so on. For example, with only one CPU and multiple programs that want to run on it, the operating system first allocates the CPU to one program, then after it has run long enough, another one gets to use the CPU, then another, and then eventually the first one again. Determining how the resource is time multiplexed—who goes next and for how long—is the task of the operating system. Another example of time multiplexing is sharing the printer. When multiple print jobs are queued up for printing on a single printer, a decision has to be made about which one is to be printed next.

The other kind of multiplexing is space multiplexing. Instead of the customers taking turns, each one gets part of the resource. For example, main memory is normally divided up among several running programs, so each one can be resident at the same time (for example, in order to take turns using the CPU). Assuming there is enough memory to hold multiple programs, it is more efficient to hold several programs in memory at once rather than give one of them all of it,

Spotify USA Inc. Exhibit 1009 Page 25

especially if it only needs a small fraction of the total. Of course, this raises issues of fairness, protection, and so on, and it is up to the operating system to solve them. Another resource that is space multiplexed is the (hard) disk. In many systems a single disk can hold files from many users at the same time. Allocating disk space and keeping track of who is using which disk blocks is a typical operating system resource management task.

1.2 HISTORY OF OPERATING SYSTEMS

Operating systems have been evolving through the years. In the following sections we will briefly look at a few of the highlights. Since operating systems have historically been closely tied to the architecture of the computers on which they run, we will look at successive generations of computers to see what their operating systems were like. This mapping of operating system generations to computer generations is crude, but it does provide some structure where there would otherwise be none.

The first true digital computer was designed by the English mathematician Charles Babbage (1792–1871). Although Babbage spent most of his life and fortune trying to build his "analytical engine," he never got it working properly because it was purely mechanical, and the technology of his day could not produce the required wheels, gears, and cogs to the high precision that he needed. Needless to say, the analytical engine did not have an operating system.

As an interesting historical aside, Babbage realized that he would need software for his analytical engine, so he hired a young woman named Ada Lovelace, who was the daughter of the famed British poet Lord Byron, as the world's first programmer. The programming language Ada® is named after her.

1.2.1 The First Generation (1945–55) Vacuum Tubes and Plugboards

After Babbage's unsuccessful efforts, little progress was made in constructing digital computers until World War II. Around the mid-1940s, Howard Aiken at Harvard, John von Neumann at the Institute for Advanced Study in Princeton, J. Presper Eckert and William Mauchley at the University of Pennsylvania, and Konrad Zuse in Germany, among others, all succeeded in building calculating engines. The first ones used mechanical relays but were very slow, with cycle times measured in seconds. Relays were later replaced by vacuum tubes. These machines were enormous, filling up entire rooms with tens of thousands of vacuum tubes, but they were still millions of times slower than even the cheapest personal computers available today.

In these early days, a single group of people designed, built, programmed, operated, and maintained each machine. All programming was done in absolute

Spotify USA Inc. Exhibit 1009 Page 26

machine language, often by wiring up plugboards to control the machine's basic functions. Programming languages were unknown (even assembly language was unknown). Operating systems were unheard of. The usual mode of operation was for the programmer to sign up for a block of time on the signup sheet on the wall, then come down to the machine room, insert his or her plugboard into the computer, and spend the next few hours hoping that none of the 20,000 or so vacuum tubes would burn out during the run. Virtually all the problems were straightforward numerical calculations, such as grinding out tables of sines, cosines, and logarithms.

By the early 1950s, the routine had improved somewhat with the introduction of punched cards. It was now possible to write programs on cards and read them in instead of using plugboards; otherwise, the procedure was the same.

1.2.2 The Second Generation (1955–65) Transistors and Batch Systems

The introduction of the transistor in the mid-1950s changed the picture radically. Computers became reliable enough that they could be manufactured and sold to paying customers with the expectation that they would continue to function long enough to get some useful work done. For the first time, there was a clear separation between designers, builders, operators, programmers, and maintenance personnel.

These machines, now called **mainframes**, were locked away in specially air conditioned computer rooms, with staffs of professional operators to run them. Only big corporations or major government agencies or universities could afford the multimillion dollar price tag. To run a **job** (i.e., a program or set of programs), a programmer would first write the program on paper (in FORTRAN or assembler), then punch it on cards. He would then bring the card deck down to the input room and hand it to one of the operators and go drink coffee until the output was ready.

When the computer finished whatever job it was currently running, an operator would go over to the printer and tear off the output and carry it over to the output room, so that the programmer could collect it later. Then he would take one of the card decks that had been brought from the input room and read it in. If the FORTRAN compiler was needed, the operator would have to get it from a file cabinet and read it in. Much computer time was wasted while operators were walking around the machine room.

Given the high cost of the equipment, it is not surprising that people quickly looked for ways to reduce the wasted time. The solution generally adopted was the **batch system**. The idea behind it was to collect a tray full of jobs in the input room and then read them onto a magnetic tape using a small (relatively) inexpensive computer, such as the IBM 1401, which was very good at reading cards, copying tapes, and printing output, but not at all good at numerical calculations.

Spotify USA Inc. Exhibit 1009 Page 27

Other, much more expensive machines, such as the IBM 7094, were used for the real computing. This situation is shown in Fig. 1-2.

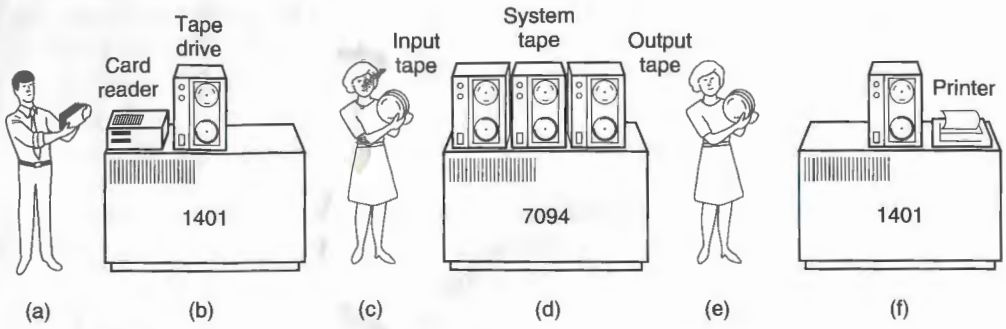


Figure 1-2. An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

After about an hour of collecting a batch of jobs, the tape was rewound and brought into the machine room, where it was mounted on a tape drive. The operator then loaded a special program (the ancestor of today's operating system), which read the first job from tape and ran it. The output was written onto a second tape, instead of being printed. After each job finished, the operating system automatically read the next job from the tape and began running it. When the whole batch was done, the operator removed the input and output tapes, replaced the input tape with the next batch, and brought the output tape to a 1401 for printing **off line** (i.e., not connected to the main computer).

The structure of a typical input job is shown in Fig. 1-3. It started out with a $JOB card, specifying the maximum run time in minutes, the account number to be charged, and the programmer's name. Then came a $FORTRAN card, telling the operating system to load the FORTRAN compiler from the system tape. It was followed by the program to be compiled, and then a $LOAD card, directing the operating system to load the object program just compiled. (Compiled programs were often written on scratch tapes and had to be loaded explicitly.) Next came the $RUN card, telling the operating system to run the program with the data following it. Finally, the $END card marked the end of the job. These primitive control cards were the forerunners of modern job control languages and command interpreters.

Large second-generation computers were used mostly for scientific and engineering calculations, such as solving the partial differential equations that often occur in physics and engineering. They were largely programmed in FORTRAN and assembly language. Typical operating systems were FMS (the Fortran Monitor System) and IBSYS, IBM's operating system for the 7094.

Spotify USA Inc. Exhibit 1009 Page 28

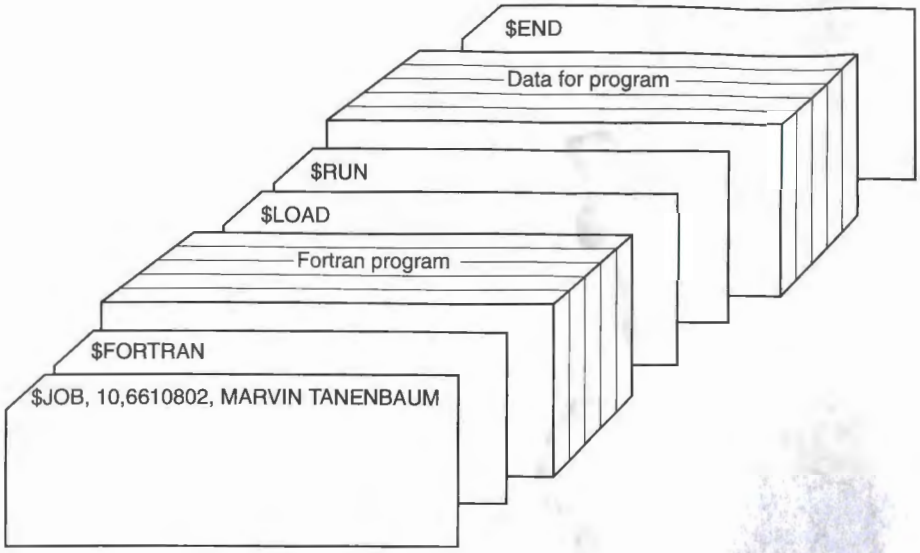


Figure 1-3. Structure of a typical FMS job.

1.2.3 The Third Generation (1965–1980) ICs and Multiprogramming

By the early 1960s, most computer manufacturers had two distinct, and totally incompatible, product lines. On the one hand there were the word-oriented, large-scale scientific computers, such as the 7094, which were used for numerical calculations in science and engineering. On the other hand, there were the character-oriented, commercial computers, such as the 1401, which were widely used for tape sorting and printing by banks and insurance companies.

Developing and maintaining two completely different product lines was an expensive proposition for the manufacturers. In addition, many new computer customers initially needed a small machine but later outgrew it and wanted a bigger machine that would run all their old programs, but faster.

IBM attempted to solve both of these problems at a single stroke by introducing the System/360. The 360 was a series of software-compatible machines ranging from 1401-sized to much more powerful than the 7094. The machines differed only in price and performance (maximum memory, processor speed, number of I/O devices permitted, and so forth). Since all the machines had the same architecture and instruction set, programs written for one machine could run on all the others, at least in theory. Furthermore, the 360 was designed to handle both scientific (i.e., numerical) and commercial computing. Thus a single family of machines could satisfy the needs of all customers. In subsequent years, IBM has come out with compatible successors to the 360 line, using more modern technology, known as the 370, 4300, 3080, and 3090 series.

Spotify USA Inc. Exhibit 1009 Page 29

The 360 was the first major computer line to use (small-scale) Integrated Circuits (ICs), thus providing a major price/performance advantage over the second-generation machines, which were built up from individual transistors. It was an immediate success, and the idea of a family of compatible computers was soon adopted by all the other major manufacturers. The descendants of these machines are still in use at computer centers today. Nowadays they are often used for managing huge databases (e.g., for airline reservation systems) or as servers for World Wide Web sites that must process thousands of requests per second.

The greatest strength of the "one family" idea was simultaneously its greatest weakness. The intention was that all software, including the operating system, **OS/360** had to work on all models. It had to run on small systems, which often just replaced 1401s for copying cards to tape, and on very large systems, which often replaced 7094s for doing weather forecasting and other heavy computing. It had to be good on systems with few peripherals and on systems with many peripherals. It had to work in commercial environments and in scientific environments. Above all, it had to be efficient for all of these different uses.

There was no way that IBM (or anybody else) could write a piece of software to meet all those conflicting requirements. The result was an enormous and extraordinarily complex operating system, probably two to three orders of magnitude larger than FMS. It consisted of millions of lines of assembly language written by thousands of programmers, and contained thousands upon thousands of bugs, which necessitated a continuous stream of new releases in an attempt to correct them. Each new release fixed some bugs and introduced new ones, so the number of bugs probably remained constant in time.

One of the designers of OS/360, Fred Brooks, subsequently wrote a witty and incisive book (Brooks, 1996) describing his experiences with OS/360. While it would be impossible to summarize the book here, suffice it to say that the cover shows a herd of prehistoric beasts stuck in a tar pit. The cover of Silberschatz et al. (2000) makes a similar point about operating systems being dinosaurs.

Despite its enormous size and problems, OS/360 and the similar third-generation operating systems produced by other computer manufacturers actually satisfied most of their customers reasonably well. They also popularized several key techniques absent in second-generation operating systems. Probably the most important of these was **multiprogramming**. On the 7094, when the current job paused to wait for a tape or other I/O operation to complete, the CPU simply sat idle until the I/O finished. With heavily CPU-bound scientific calculations, I/O is infrequent, so this wasted time is not significant. With commercial data processing, the I/O wait time can often be 80 or 90 percent of the total time, so something had to be done to avoid having the (expensive) CPU be idle so much.

The solution that evolved was to partition memory into several pieces, with a different job in each partition, as shown in Fig. 1-4. While one job was waiting for I/O to complete, another job could be using the CPU. If enough jobs could be held in main memory at once, the CPU could be kept busy nearly 100 percent of

Spotify USA Inc. Exhibit 1009 Page 30

the time. Having multiple jobs safely in memory at once requires special hardware to protect each job against snooping and mischief by the other ones, but the 360 and other third-generation systems were equipped with this hardware.

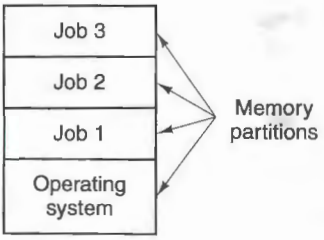


Figure 1-4. A multiprogramming system with three jobs in memory.

Another major feature present in third-generation operating systems was the ability to read jobs from cards onto the disk as soon as they were brought to the computer room. Then, whenever a running job finished, the operating system could load a new job from the disk into the now-empty partition and run it. This technique is called **spooling** (from Simultaneous Peripheral Operation On Line) and was also used for output. With spooling, the 1401s were no longer needed, and much carrying of tapes disappeared.

Although third-generation operating systems were well suited for big scientific calculations and massive commercial data processing runs, they were still basically batch systems. Many programmers pined for the first-generation days when they had the machine all to themselves for a few hours, so they could debug their programs quickly. With third-generation systems, the time between submitting a job and getting back the output was often several hours, so a single misplaced comma could cause a compilation to fail, and the programmer to waste half a day.

This desire for quick response time paved the way for **timesharing**, a variant of multiprogramming, in which each user has an online terminal. In a timesharing system, if 20 users are logged in and 17 of them are thinking or talking or drinking coffee, the CPU can be allocated in turn to the three jobs that want service. Since people debugging programs usually issue short commands (e.g., compile a five-page procedure†) rather than long ones (e.g., sort a million-record file), the computer can provide fast, interactive service to a number of users and perhaps also work on big batch jobs in the background when the CPU is otherwise idle. The first serious timesharing system, **CTSS (Compatible Time Sharing System)**, was developed at M.I.T. on a specially modified 7094 (Corbató et al., 1962). However, timesharing did not really become popular until the necessary protection hardware became widespread during the third generation.

†We will use the terms "procedure," "subroutine," and "function" interchangeably in this book.

Spotify USA Inc. Exhibit 1009 Page 31

After the success of the CTSS system, MIT, Bell Labs, and General Electric (then a major computer manufacturer) decided to embark on the development of a "computer utility," a machine that would support hundreds of simultaneous timesharing users. Their model was the electricity distribution system—when you need electric power, you just stick a plug in the wall, and within reason, as much power as you need will be there. The designers of this system, known as **MULTICS** (**MULTiplexed Information and Computing Service**), envisioned one huge machine providing computing power for everyone in the Boston area. The idea that machines far more powerful than their GE-645 mainframe would be sold for a thousand dollars by the millions only 30 years later was pure science fiction. Sort of like the idea of supersonic trans-Atlantic undersea trains now.

MULTICS was a mixed success. It was designed to support hundreds of users on a machine only slightly more powerful than an Intel 386-based PC, although it had much more I/O capacity. This is not quite as crazy as it sounds, since people knew how to write small, efficient programs in those days, a skill that has subsequently been lost. There were many reasons that MULTICS did not take over the world, not the least of which is that it was written in PL/I, and the PL/I compiler was years late and barely worked at all when it finally arrived. In addition, MULTICS was enormously ambitious for its time, much like Charles Babbage's analytical engine in the nineteenth century.

To make a long story short, MULTICS introduced many seminal ideas into the computer literature, but turning it into a serious product and a major commercial success was a lot harder than anyone had expected. Bell Labs dropped out of the project, and General Electric quit the computer business altogether. However, M.I.T. persisted and eventually got MULTICS working. It was ultimately sold as a commercial product by the company that bought GE's computer business (Honeywell) and installed by about 80 major companies and universities worldwide. While their numbers were small, MULTICS users were fiercely loyal. General Motors, Ford, and the U.S. National Security Agency, for example, only shut down their MULTICS systems in the late 1990s, 30 years after MULTICS was released.

For the moment, the concept of a computer utility has fizzled out but it may well come back in the form of massive centralized Internet servers to which relatively dumb user machines are attached, with most of the work happening on the big servers. The motivation here is likely to be that most people do not want to administrate an increasingly complex and finicky computer system and would prefer to have that work done by a team of professionals working for the company running the server. E-commerce is already evolving in this direction, with various companies running e-malls on multiprocessor servers to which simple client machines connect, very much in the spirit of the MULTICS design.

Despite its lack of commercial success, MULTICS had a huge influence on subsequent operating systems. It is described in (Corbató et al., 1972; Corbató and Vyssotsky, 1965; Daley and Dennis, 1968; Organick, 1972; and Saltzer, 1974). It

Spotify USA Inc. Exhibit 1009 Page 32

also has a still-active Web site, *www.multicians.org*, with a great deal of information about the system, its designers, and its users.

Another major development during the third generation was the phenomenal growth of minicomputers, starting with the DEC PDP-1 in 1961. The PDP-1 had only 4K of 18-bit words, but at $120,000 per machine (less than 5 percent of the price of a 7094), it sold like hotcakes. For certain kinds of nonnumerical work, it was almost as fast as the 7094 and gave birth to a whole new industry. It was quickly followed by a series of other PDPs (unlike IBM's family, all incompatible) culminating in the PDP-11.

One of the computer scientists at Bell Labs who had worked on the MULTICS project, Ken Thompson, subsequently found a small PDP-7 minicomputer that no one was using and set out to write a stripped-down, one-user version of MULTICS. This work later developed into the **UNIX**® operating system, which became popular in the academic world, with government agencies, and with many companies.

The history of UNIX has been told elsewhere (e.g., Salus, 1994). Part of that story will be given in Chap. 10. For now, suffice it to say, that because the source code was widely available, various organizations developed their own (incompatible) versions, which led to chaos. Two major versions developed, **System V**, from AT&T, and **BSD**, (Berkeley Software Distribution) from the University of California at Berkeley. These had minor variants as well. To make it possible to write programs that could run on any UNIX system, IEEE developed a standard for UNIX, called **POSIX**, that most versions of UNIX now support. POSIX defines a minimal system call interface that conformant UNIX systems must support. In fact, some other operating systems now also support the POSIX interface.

As an aside, it is worth mentioning that in 1987, the author released a small clone of UNIX, called **MINIX,** for educational purposes. Functionally, MINIX is very similar to UNIX, including POSIX support. A book describing its internal operation and listing the source code in an appendix is also available (Tanenbaum and Woodhull, 1997). MINIX is available for free (including all the source code) over the Internet at URL *www.cs.vu.nl/~ast/minix.html*.

The desire for a free production (as opposed to educational) version of MINIX led a Finnish student, Linus Torvalds, to write **Linux**. This system was developed on MINIX and originally supported various MINIX features (e.g., the MINIX file system). It has since been extended in many ways but still retains a large amount of underlying structure common to MINIX, and to UNIX (upon which the former was based). Most of what will be said about UNIX in this book thus applies to System V, BSD, MINIX, Linux, and other versions and clones of UNIX as well.

1.2.4 The Fourth Generation (1980–Present) Personal Computers

With the development of LSI (Large Scale Integration) circuits, chips containing thousands of transistors on a square centimeter of silicon, the age of the personal computer dawned. In terms of architecture, personal computers (initially

Spotify USA Inc. Exhibit 1009 Page 33

called **microcomputers**) were not all that different from minicomputers of the PDP-11 class, but in terms of price they certainly were different. Where the minicomputer made it possible for a department in a company or university to have its own computer, the microprocessor chip made it possible for a single individual to have his or her own personal computer.

In 1974, when Intel came out with the 8080, the first general-purpose 8-bit CPU, it wanted an operating system for the 8080, in part to be able to test it. Intel asked one of its consultants, Gary Kildall, to write one. Kildall and a friend first built a controller for the newly-released Shugart Associates 8-inch floppy disk and hooked the floppy disk up to the 8080, thus producing the first microcomputer with a disk. Kildall then wrote a disk-based operating system called **CP/M** (**Control Program for Microcomputers**) for it. Since Intel did not think that disk-based microcomputers had much of a future, when Kildall asked for the rights to CP/M, Intel granted his request. Kildall then formed a company, Digital Research, to further develop and sell CP/M.

In 1977, Digital Research rewrote CP/M to make it suitable for running on the many microcomputers using the 8080, Zilog Z80, and other CPU chips. Many application programs were written to run on CP/M, allowing it to completely dominate the world of microcomputing for about 5 years.

In the early 1980s, IBM designed the IBM PC and looked around for software to run on it. People from IBM contacted Bill Gates to license his BASIC interpreter. They also asked him if he knew of an operating system to run on the PC. Gates suggested that IBM contact Digital Research, then the world's dominant operating systems company. Making what was surely the worst business decision in recorded history, Kildall refused to meet with IBM, sending a subordinate instead. To make matters worse, his lawyer even refused to sign IBM's nondisclosure agreement covering the not-yet-announced PC. Consequently, IBM went back to Gates asking if he could provide them with an operating system.

When IBM came back, Gates realized that a local computer manufacturer, Seattle Computer Products, had a suitable operating system, **DOS** (**Disk Operating System**). He approached them and asked to buy it (allegedly for $50,000), which they readily accepted. Gates then offered IBM a DOS/BASIC package, which IBM accepted. IBM wanted certain modifications, so Gates hired the person who wrote DOS, Tim Paterson, as an employee of Gates' fledgling company, Microsoft, to make them. The revised system was renamed **MS-DOS** (**MicroSoft Disk Operating System**) and quickly came to dominate the IBM PC market. A key factor here was Gates' (in retrospect, extremely wise) decision to sell MS-DOS to computer companies for bundling with their hardware, compared to Kildall's attempt to sell CP/M to end users one at a time (at least initially).

By the time the IBM PC/AT came out in 1983 with the Intel 80286 CPU, MS-DOS was firmly entrenched and CP/M was on its last legs. MS-DOS was later widely used on the 80386 and 80486. Although the initial version of MS-DOS was fairly primitive, subsequent versions included more advanced features, including

Spotify USA Inc. Exhibit 1009 Page 34

many taken from UNIX. (Microsoft was well aware of UNIX, even selling a microcomputer version of it called XENIX during the company's early years.)

CP/M, MS-DOS, and other operating systems for early microcomputers were all based on users typing in commands from the keyboard. That eventually changed due to research done by Doug Engelbart at Stanford Research Institute in the 1960s. Engelbart invented the **GUI** (**Graphical User Interface**), pronounced "gooey," complete with windows, icons, menus, and mouse. These ideas were adopted by researchers at Xerox PARC and incorporated into machines they built.

One day, Steve Jobs, who co-invented the Apple computer in his garage, visited PARC, saw a GUI, and instantly realized its potential value, something Xerox management famously did not (Smith and Alexander, 1988). Jobs then embarked on building an Apple with a GUI. This project led to the Lisa, which was too expensive and failed commercially. Jobs' second attempt, the Apple Macintosh, was a huge success, not only because it was much cheaper than the Lisa, but also because it was **user friendly**, meaning that it was intended for users who not only knew nothing about computers but furthermore had absolutely no intention whatsoever of learning.

When Microsoft decided to build a successor to MS-DOS, it was strongly influenced by the success of the Macintosh. It produced a GUI-based system called Windows, which originally ran on top of MS-DOS (i.e., it was more like a shell than a true operating system). For about 10 years, from 1985 to 1995, Windows was just a graphical environment on top of MS-DOS. However, starting in 1995 a freestanding version of Windows, Windows 95, was released that incorporated many operating system features into it, using the underlying MS-DOS system only for booting and running old MS-DOS programs. In 1998, a slightly modified version of this system, called Windows 98 was released. Nevertheless, both Windows 95 and Windows 98 still contain a large amount of 16-bit Intel assembly language.

Another Microsoft operating system is **Windows NT** (NT stands for New Technology), which is compatible with Windows 95 at a certain level, but a complete rewrite from scratch internally. It is a full 32-bit system. The lead designer for Windows NT was David Cutler, who was also one of the designers of the VAX VMS operating system, so some ideas from VMS are present in NT. Microsoft expected that the first version of NT would kill off MS-DOS and all other versions of Windows since it was a vastly superior system, but it fizzled. Only with Windows NT 4.0 did it finally catch on in a big way, especially on corporate networks. Version 5 of Windows NT was renamed Windows 2000 in early 1999. It was intended to be the successor to both Windows 98 and Windows NT 4.0. That did not quite work out either, so Microsoft came out with yet another version of Windows 98 called **Windows Me** (**Millennium edition**).

The other major contender in the personal computer world is UNIX (and its various derivatives). UNIX is strongest on workstations and other high-end computers, such as network servers. It is especially popular on machines powered by

Spotify USA Inc. Exhibit 1009 Page 35

high-performance RISC chips. On Pentium-based computers, Linux is becoming a popular alternative to Windows for students and increasingly many corporate users. (As an aside, throughout this book we will use the term "Pentium" to mean the Pentium I, II, III, and 4.)

Although many UNIX users, especially experienced programmers, prefer a command-based interface to a GUI, nearly all UNIX systems support a windowing system called the **X Windows** system produced at M.I.T. This system handles the basic window management, allowing users to create, delete, move, and resize windows using a mouse. Often a complete GUI, such as **Motif**, is available to run on top of the X Windows system giving UNIX a look and feel something like the Macintosh or Microsoft Windows, for those UNIX users who want such a thing.

An interesting development that began taking place during the mid-1980s is the growth of networks of personal computers running **network operating systems** and **distributed operating systems** (Tanenbaum and Van Steen, 2002). In a network operating system, the users are aware of the existence of multiple computers and can log in to remote machines and copy files from one machine to another. Each machine runs its own local operating system and has its own local user (or users).

Network operating systems are not fundamentally different from single-processor operating systems. They obviously need a network interface controller and some low-level software to drive it, as well as programs to achieve remote login and remote file access, but these additions do not change the essential structure of the operating system.

A distributed operating system, in contrast, is one that appears to its users as a traditional uniprocessor system, even though it is actually composed of multiple processors. The users should not be aware of where their programs are being run or where their files are located; that should all be handled automatically and efficiently by the operating system.

True distributed operating systems require more than just adding a little code to a uniprocessor operating system, because distributed and centralized systems differ in critical ways. Distributed systems, for example, often allow applications to run on several processors at the same time, thus requiring more complex processor scheduling algorithms in order to optimize the amount of parallelism.

Communication delays within the network often mean that these (and other) algorithms must run with incomplete, outdated, or even incorrect information. This situation is radically different from a single-processor system in which the operating system has complete information about the system state.

1.2.5 Ontogeny Recapitulates Phylogeny

After Charles Darwin's book *The Origin of the Species* was published, the German zoologist Ernst Haeckel stated that "Ontogeny Recapitulates Phylogeny." By this he meant that the development of an embryo (ontogeny) repeats

Spotify USA Inc. Exhibit 1009 Page 36

(i.e., recapitulates) the evolution of the species (phylogeny). In other words, after fertilization, a human egg goes through stages of being a fish, a pig, and so on before turning into a human baby. Modern biologists regard this as a gross simplification, but it still has a kernel of truth in it.

Something analogous has happened in the computer industry. Each new species (mainframe, minicomputer, personal computer, embedded computer, smart card, etc.) seems to go through the development that its ancestors did. The first mainframes were programmed entirely in assembly language. Even complex programs, like compilers and operating systems, were written in assembler. By the time minicomputers appeared on the scene, FORTRAN, COBOL, and other high-level languages were common on mainframes, but the new minicomputers were nevertheless programmed in assembler (for lack of memory). When microcomputers (early personal computers) were invented, they, too, were programmed in assembler, even though by then minicomputers were also programmed in high-level languages. Palmtop computers also started with assembly code but quickly moved on to high-level languages (mostly because the development work was done on bigger machines). The same is true for smart cards.

Now let us look at operating systems. The first mainframes initially had no protection hardware and no support for multiprogramming, so they ran simple operating systems that handled one manually-loaded program at a time. Later they acquired the hardware and operating system support to handle multiple programs at once, and then full timesharing capabilities.

When minicomputers first appeared, they also had no protection hardware and ran one manually-loaded program at a time, even though multiprogramming was well established in the mainframe world by then. Gradually, they acquired protection hardware and the ability to run two or more programs at once. The first microcomputers were also capable of running only one program at a time, but later acquired the ability to multiprogram. Palmtops and smart cards went the same route.

Disks first appeared on large mainframes, then on minicomputers, microcomputers, and so on down the line. Even now, smart cards do not have hard disks, but with the advent of flash ROM, they will soon have the equivalent of it. When disks first appeared, primitive file systems sprung up. On the CDC 6600, easily the most powerful mainframe in the world during much of the 1960s, the file system consisted of users having the ability to create a file and then declare it to be permanent, meaning it stayed on the disk even after the creating program exited. To access such a file later, a program had to attach it with a special command and give its password (supplied when the file was made permanent). In effect, there was a single directory shared by all users. It was up to the users to avoid file name conflicts. Early minicomputer file systems had a single directory shared by all users and so did early microcomputer file systems.

Virtual memory (the ability to run programs larger than the physical memory) had a similar development. It first appeared in mainframes, minicomputers,

Spotify USA Inc. Exhibit 1009 Page 37

microcomputers and gradually worked its way down to smaller and smaller systems. Networking had a similar history.

In all cases, the software development was dictated by the technology. The first microcomputers, for example, had something like 4 KB of memory and no protection hardware. High-level languages and multiprogramming were simply too much for such a tiny system to handle. As the microcomputers evolved into modern personal computers, they acquired the necessary hardware and then the necessary software to handle more advanced features. It is likely that this development will continue for years to come. Other fields may also have this wheel of reincarnation, but in the computer industry it seems to spin faster.

1.3 THE OPERATING SYSTEM ZOO

All of this history and development has left us with a wide variety of operating systems, not all of which are widely known. In this section we will briefly touch upon seven of them. We will come back to some of these different kinds of systems later in the book.

1.3.1 Mainframe Operating Systems

At the high end are the operating systems for the mainframes, those room-sized computers still found in major corporate data centers. These computers distinguish themselves from personal computers in terms of their I/O capacity. A mainframe with 1000 disks and thousands of gigabytes of data is not unusual; a personal computer with these specifications would be odd indeed. Mainframes are also making something of a comeback as high-end Web servers, servers for large-scale electronic commerce sites, and servers for business-to-business transactions.

The operating systems for mainframes are heavily oriented toward processing many jobs at once, most of which need prodigious amounts of I/O. They typically offer three kinds of services: batch, transaction processing, and timesharing. A batch system is one that processes routine jobs without any interactive user present. Claims processing in an insurance company or sales reporting for a chain of stores is typically done in batch mode. Transaction processing systems handle large numbers of small requests, for example, check processing at a bank or airline reservations. Each unit of work is small, but the system must handle hundreds or thousands per second. Timesharing systems allow multiple remote users to run jobs on the computer at once, such as querying a big database. These functions are closely related; mainframe operating systems often perform all of them. An example mainframe operating system is OS/390, a descendant of OS/360.

Spotify USA Inc. Exhibit 1009 Page 38

1.3.2 Server Operating Systems

One level down are the server operating systems. They run on servers, which are either very large personal computers, workstations, or even mainframes. They serve multiple users at once over a network and allow the users to share hardware and software resources. Servers can provide print service, file service, or Web service. Internet providers run many server machines to support their customers and Web sites use servers to store the Web pages and handle the incoming requests. Typical server operating systems are UNIX and Windows 2000. Linux is also gaining ground for servers.

1.3.3 Multiprocessor Operating Systems

An increasingly common way to get major-league computing power is to connect multiple CPUs into a single system. Depending on precisely how they are connected and what is shared, these systems are called parallel computers, multicomputers, or multiprocessors. They need special operating systems, but often these are variations on the server operating systems, with special features for communication and connectivity.

1.3.4 Personal Computer Operating Systems

The next category is the personal computer operating system. Their job is to provide a good interface to a single user. They are widely used for word processing, spreadsheets, and Internet access. Common examples are Windows 98, Windows 2000, the Macintosh operating system, and Linux. Personal computer operating systems are so widely known that probably little introduction is needed. In fact, many people are not even aware that other kinds exist.

1.3.5 Real-Time Operating Systems

Another type of operating system is the real-time system. These systems are characterized by having time as a key parameter. For example, in industrial process control systems, real-time computers have to collect data about the production process and use it to control machines in the factory. Often there are hard deadlines that must be met. For example, if a car is moving down an assembly line, certain actions must take place at certain instants of time. If a welding robot welds too early or too late, the car will be ruined. If the action absolutely *must* occur at a certain moment (or within a certain range), we have a **hard real-time system**.

Another kind of real-time system is a **soft real-time system**, in which missing an occasional deadline is acceptable. Digital audio or multimedia systems fall in this category. VxWorks and QNX are well-known real-time operating systems.

Spotify USA Inc. Exhibit 1009 Page 39

1.3.6 Embedded Operating Systems

Continuing on down to smaller and smaller systems, we come to palmtop computers and embedded systems. A palmtop computer or **PDA (Personal Digital Assistant)** is a small computer that fits in a shirt pocket and performs a small number of functions such as an electronic address book and memo pad. Embedded systems run on the computers that control devices that are not generally thought of as computers, such as TV sets, microwave ovens, and mobile telephones. These often have some characteristics of real-time systems but also have size, memory, and power restrictions that make them special. Examples of such operating systems are PalmOS and Windows CE (Consumer Electronics).

1.3.7 Smart Card Operating Systems

The smallest operating systems run on smart cards, which are credit card-sized devices containing a CPU chip. They have very severe processing power and memory constraints. Some of them can handle only a single function, such as electronic payments, but others can handle multiple functions on the same smart card. Often these are proprietary systems.

Some smart cards are Java oriented. What this means is that the ROM on the smart card holds an interpreter for the Java Virtual Machine (JVM). Java applets (small programs) are downloaded to the card and are interpreted by the JVM interpreter. Some of these cards can handle multiple Java applets at the same time, leading to multiprogramming and the need to schedule them. Resource management and protection also become an issue when two or more applets are present at the same time. These issues must be handled by the (usually extremely primitive) operating system present on the card.

1.4 COMPUTER HARDWARE REVIEW

An operating system is intimately tied to the hardware of the computer it runs on. It extends the computer's instruction set and manages its resources. To work, it must know a great deal about the hardware, at least, about how the hardware appears to the programmer.

Conceptually, a simple personal computer can be abstracted to a model resembling that of Fig. 1-5. The CPU, memory, and I/O devices are all connected by a system bus and communicate with one another over it. Modern personal computers have a more complicated structure, involving multiple buses, which we will look at later. For the time being, this model will be sufficient. In the following sections, we will briefly review these components and examine some of the hardware issues that are of concern to operating system designers.

Spotify USA Inc. Exhibit 1009 Page 40

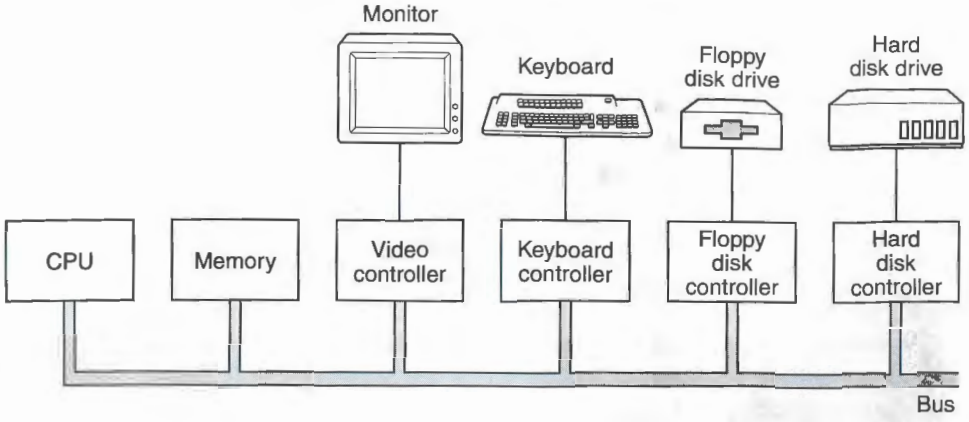


Figure 1-5. Some of the components of a simple personal computer.

1.4.1 Processors

The "brain" of the computer is the CPU. It fetches instructions from memory and executes them. The basic cycle of every CPU is to fetch the first instruction from memory, decode it to determine its type and operands, execute it, and then fetch, decode, and execute subsequent instructions. In this way, programs are carried out.

Each CPU has a specific set of instructions that it can execute. Thus a Pentium cannot execute SPARC programs and a SPARC cannot execute Pentium programs. Because accessing memory to get an instruction or data word takes much longer than executing an instruction, all CPUs contain some registers inside to hold key variables and temporary results. Thus the instruction set generally contains instructions to load a word from memory into a register, and store a word from a register into memory. Other instructions combine two operands from registers, memory, or both into a result, such as adding two words and storing the result in a register or in memory.

In addition to the general registers used to hold variables and temporary results, most computers have several special registers that are visible to the programmer. One of these is the **program counter**, which contains the memory address of the next instruction to be fetched. After that instruction has been fetched, the program counter is updated to point to its successor.

Another register is the **stack pointer**, which points to the top of the current stack in memory. The stack contains one frame for each procedure that has been entered but not yet exited. A procedure's stack frame holds those input parameters, local variables, and temporary variables that are not kept in registers.

Yet another register is the **PSW** (**Program Status Word**). This register contains the condition code bits, which are set by comparison instructions, the CPU priority, the mode (user or kernel), and various other control bits. User programs

Spotify USA Inc. Exhibit 1009 Page 41

may normally read the entire PSW but typically may write only some of its fields. The PSW plays an important role in system calls and I/O.

The operating system must be aware of all the registers. When time multiplexing the CPU, the operating system will often stop the running program to (re)start another one. Every time it stops a running program, the operating system must save all the registers so they can be restored when the program runs later.

To improve performance, CPU designers have long abandoned the simple model of fetching, decoding, and executing one instruction at a time. Many modern CPUs have facilities for executing more than one instruction at the same time. For example, a CPU might have separate fetch, decode, and execute units, so that while it was executing instruction n, it could also be decoding instruction $n+1$ and fetching instruction $n+2$. Such an organization is called a **pipeline** and is illustrated in Fig. 1-6(a) for a pipeline with three stages. Longer pipelines are common. In most pipeline designs, once an instruction has been fetched into the pipeline, it must be executed, even if the preceding instruction was a conditional branch that was taken. Pipelines cause compiler writers and operating system writers great headaches because they expose the complexities of the underlying machine to them.

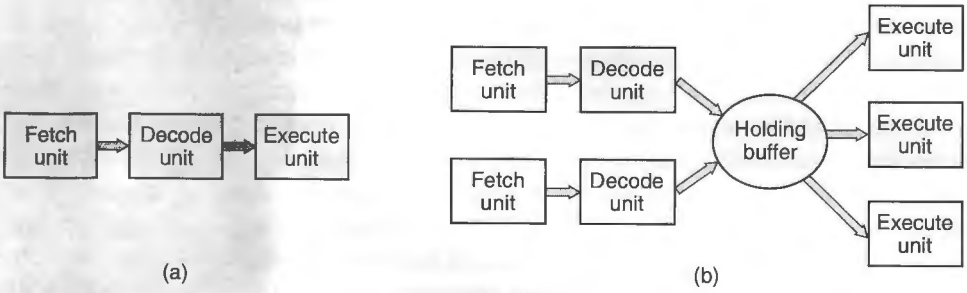


Figure 1-6. (a) A three-stage pipeline. (b) A superscalar CPU.

Even more advanced than a pipeline design is a **superscalar** CPU, shown in Fig. 1-6(b). In this design, multiple execution units are present, for example, one for integer arithmetic, one for floating-point arithmetic, and one for Boolean operations. Two or more instructions are fetched at once, decoded, and dumped into a holding buffer until they can be executed. As soon as an execution unit is free, it looks in the holding buffer to see if there is an instruction it can handle, and if so, it removes the instruction from the buffer and executes it. An implication of this design is that program instructions are often executed out of order. For the most part, it is up to the hardware to make sure the result produced is the same one a sequential implementation would have produced, but an annoying amount of the complexity is foisted onto the operating system, as we shall see.

Most CPUs, except very simple ones used in embedded systems, have two modes, kernel mode and user mode, as mentioned earlier. Usually a bit in the

Spotify USA Inc. Exhibit 1009 Page 42

PSW controls the mode. When running in kernel mode, the CPU can execute every instruction in its instruction set and use every feature of the hardware. The operating system runs in kernel mode, giving it access to the complete hardware.

In contrast, user programs run in user mode, which permits only a subset of the instructions to be executed and a subset of the features to be accessed. Generally, all instructions involving I/O and memory protection are disallowed in user mode. Setting the PSW mode bit to kernel mode is also forbidden, of course.

To obtain services from the operating system, a user program must make a **system call**, which traps into the kernel and invokes the operating system. The TRAP instruction switches from user mode to kernel mode and starts the operating system. When the work has been completed, control is returned to the user program at the instruction following the system call. We will explain the details of the system call process later in this chapter. As a note on typography, we will use the lower case Helvetica font to indicate system calls in running text, like this: read.

It is worth noting that computers have traps other than the instruction for executing a system call. Most of the other traps are caused by the hardware to warn of an exceptional situation such as an attempt to divide by 0 or a floating-point underflow. In all cases the operating system gets control and must decide what to do. Sometimes the program must be terminated with an error. Other times the error can be ignored (an underflowed number can be set to 0). Finally, when the program has announced in advance that it wants to handle certain kinds of conditions, control can be passed back to the program to let it deal with the problem.

1.4.2 Memory

The second major component in any computer is the memory. Ideally, a memory should be extremely fast (faster than executing an instruction so the CPU is not held up by the memory), abundantly large, and dirt cheap. No current technology satisfies all of these goals, so a different approach is taken. The memory system is constructed as a hierarchy of layers, as shown in Fig. 1-7.

The top layer consists of the registers internal to the CPU. They are made of the same material as the CPU and are thus just as fast as the CPU. Consequently, there is no delay in accessing them. The storage capacity available in them is typically 32 × 32-bits on a 32-bit CPU and 64 × 64-bits on a 64-bit CPU. Less than 1 KB in both cases. Programs must manage the registers (i.e., decide what to keep in them) themselves, in software.

Next comes the cache memory, which is mostly controlled by the hardware. Main memory is divided up into **cache lines**, typically 64 bytes, with addresses 0 to 63 in cache line 0, addresses 64 to 127 in cache line 1, and so on. The most heavily used cache lines are kept in a high-speed cache located inside or very close to the CPU. When the program needs to read a memory word, the cache hardware checks to see if the line needed is in the cache. If it is, called a **cache**

Spotify USA Inc. Exhibit 1009 Page 43

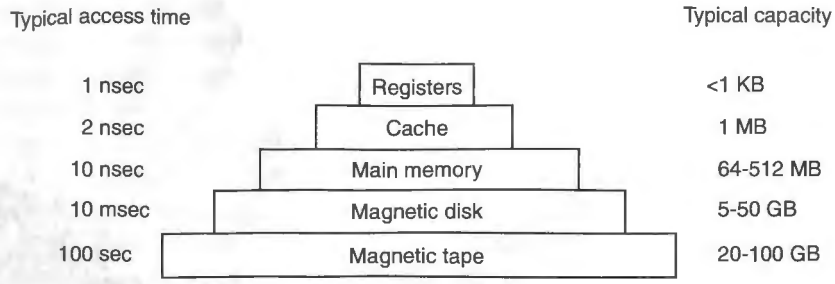


Figure 1-7. A typical memory hierarchy. The numbers are very rough approximations.

hit, the request is satisfied from the cache and no memory request is sent over the bus to the main memory. Cache hits normally take about two clock cycles. Cache misses have to go to memory, with a substantial time penalty. Cache memory is limited in size due to its high cost. Some machines have two or even three levels of cache, each one slower and bigger than the one before it.

Main memory comes next. This is the workhorse of the memory system. Main memory is often called **RAM** (**Random Access Memory**). Old timers sometimes call it **core memory**, because computers in the 1950s and 1960s used tiny magnetizable ferrite cores for main memory. Currently, memories are tens to hundreds of megabytes and growing rapidly. All CPU requests that cannot be satisfied out of the cache go to main memory.

Next in the hierarchy is magnetic disk (hard disk). Disk storage is two orders of magnitude cheaper than RAM per bit and often two orders of magnitude larger as well. The only problem is that the time to randomly access data on it is close to three orders of magnitude slower. This low speed is due to the fact that a disk is a mechanical device, as shown in Fig. 1-8.

A disk consists of one or more metal platters that rotate at 5400, 7200, or 10,800 rpm A mechanical arm pivots over the platters from the corner, similar to the pickup arm on an old 33 rpm phonograph for playing vinyl records. Information is written onto the disk in a series of concentric circles. At any given arm position, each of the heads can read an annular region called a **track**. Together, all the tracks for a given arm position form a **cylinder**.

Each track is divided into some number of sectors, typically 512 bytes per sector. On modern disks, the outer cylinders contain more sectors than the inner ones. Moving the arm from one cylinder to the next one takes about 1 msec. Moving it to a random cylinder typically takes 5 msec to 10 msec, depending on the drive. Once the arm is on the correct track, the drive must wait for the needed sector to rotate under the head, an additional delay of 5 msec to 10 msec, depending on the drive's rpm. Once the sector is under the head, reading or writing occurs at a rate of 5 MB/sec on low-end disks to 160 MB/sec on faster ones.

Spotify USA Inc. Exhibit 1009 Page 44

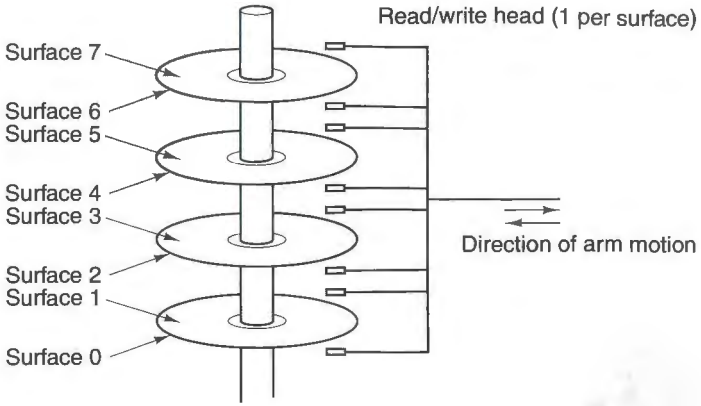


Figure 1-8. Structure of a disk drive.

The final layer in the memory hierarchy is magnetic tape. This medium is often used as a backup for disk storage and for holding very large data sets. To access a tape, it must first be put into a tape reader, either by a person or a robot (automated tape handling is common at installations with huge databases). Then the tape may have to be spooled forwarded to get to the requested block. All in all, this could take minutes. The big plus of tape is that it is exceedingly cheap per bit and removable, which is important for backup tapes that must be stored off-site in order to survive fires, floods, earthquakes, etc.

The memory hierarchy we have discussed is typical, but some installations do not have all the layers or have a few different ones (such as optical disk). Still, in all of them, as one goes down the hierarchy, the random access time increases dramatically, the capacity increases equally dramatically, and the cost per bit drops enormously. Consequently, it is likely that memory hierarchies will be around for years to come.

In addition to the kinds of memory discussed above, many computers have a small amount of nonvolatile random access memory. Unlike RAM, nonvolatile memory does not lose its contents when the power is switched off. **ROM (Read Only Memory)** is programmed at the factory and cannot be changed afterward. It is fast and inexpensive. On some computers, the bootstrap loader used to start the computer is contained in ROM. Also, some I/O cards come with ROM for handling low-level device control.

EEPROM (Electrically Erasable ROM) and **flash RAM** are also nonvolatile, but in contrast to ROM can be erased and rewritten. However, writing them takes orders of magnitude more time than writing RAM, so they are used in the same way ROM is, only with the additional feature that it is now possible to correct bugs in programs they hold by rewriting them in the field.

Yet another kind of memory is CMOS, which is volatile. Many computers use CMOS memory to hold the current time and date. The CMOS memory and

Spotify USA Inc. Exhibit 1009 Page 45

the clock circuit that increments the time in it are powered by a small battery, so the time is correctly updated, even when the computer is unplugged. The CMOS memory can also hold the configuration parameters, such as which disk to boot from. CMOS is used because it draws so little power that the original factory-installed battery often lasts for several years. However, when it begins to fail, the computer can appear to have Alzheimer's disease, forgetting things that it has known for years, like which hard disk to boot from.

Let us now focus on main memory for a little while. It is often desirable to hold multiple programs in memory at once. If one program is blocked waiting for a disk read to complete, another program can use the CPU, giving a better CPU utilization. However, with two or more programs in main memory at once, two problems must be solved:

1. How to protect the programs from one another and the kernel from them all.
2. How to handle relocation.

Many solutions are possible. However, all of them involve equipping the CPU with special hardware.

The first problem is obvious, but the second one is a bit more subtle. When a program is compiled and linked, the compiler and linker do not know where in physical memory it will be loaded when it is executed. For this reason, they usually assume it will start at address 0, and just put the first instruction there. Suppose that the first instruction fetches a word from memory address 10,000. Now suppose that the entire program and data are loaded starting at address 50,000. When the first instruction executes, it will fail because it will reference the word at 10,000, instead of the word at 60,000. To solve this problem, we need to either relocate the program at load time, finding all the addresses and modifying them (which is doable, but expensive), or have relocation done on-the-fly during execution.

The simplest solution is shown in Fig. 1-9(a). In this figure we see a computer equipped with two special registers, the **base register** and the **limit register**. (Please note that in this book, numbers beginning with 0x are in hexadecimal—the C language convention. Similarly, numbers beginning with a leading zero are in octal.) When a program is run, the base register is set to point to the start of its program text and the limit register tells how large the combined program text and data are. When an instruction is to be fetched, the hardware checks to see if the program counter is less than the limit register, and if it is, adds it to the base register and sends the sum to memory. Similarly, when the program wants to fetch a data word (e.g., from address 10,000), the hardware automatically adds the contents of the base register (e.g., 50,000) to that address and sends the sum (60,000) to the memory. The base register makes it impossible for a program to reference any part of memory below itself. Furthermore, the limit register makes it impos-

Spotify USA Inc. Exhibit 1009 Page 46

sible to reference any part of memory above itself. Thus this scheme solves both the protection and the relocation problem at the cost of two new registers and a slight increase in cycle time (to perform the limit check and addition).

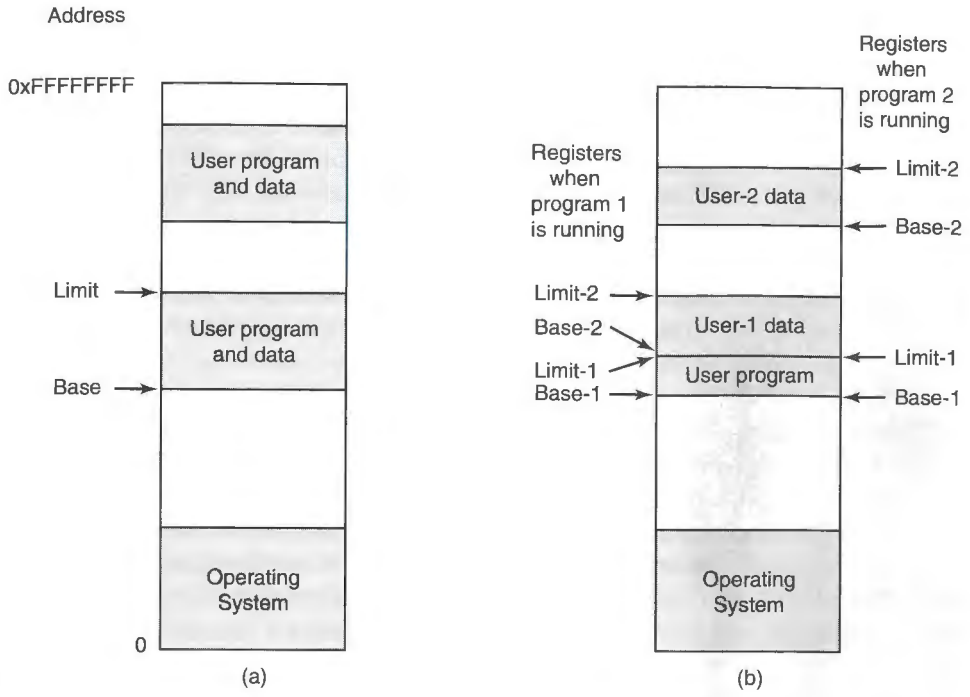


Figure 1-9. (a) Use of one base-limit pair. The program can access memory between the base and the limit. (b) Use of two base-limit pairs. The program code is between Base-1 and Limit-1 whereas the data are between Base-2 and Limit-2.

The check and mapping result in converting an address generated by the program, called a **virtual address**, into an address used by the memory, called a **physical address**. The device that performs the check and mapping is called the **MMU (Memory Management Unit)**. It is located on the CPU chip or close to it, but is logically between the CPU and the memory.

A more sophisticated MMU is illustrated in Fig. 1-9(b). Here we have an MMU with two pairs of base and limit registers, one for the program text and one for the data. The program counter and all other references to the program text use pair 1 and data references use pair 2. As a consequence, it is now possible to have multiple users share the same program with only one copy of it in memory, something not possible with the first scheme. When program 1 is running, the four registers are set as indicated by the arrows to the left of Fig. 1-9(b). When program 2 is running, they are set as indicated by the arrows to the right of the figure. Much more sophisticated MMUs exist. We will study some of them later in this

Spotify USA Inc. Exhibit 1009 Page 47

book. The thing to note here is that managing the MMU must be an operating system function, since users cannot be trusted to do it correctly.

Two aspects of the memory system have a major effect on performance. First, caches hide the relatively slow speed of memory. When a program has been running for a while, the cache is full of that program's cache lines, giving good performance. However, when the operating system switches from one program to another, the cache remains full of the first program's cache lines. The ones needed by the new program must be loaded one at a time from physical memory. This operation can be a major performance hit if it happens too often.

Second, when switching from one program to another, the MMU registers have to be changed. In Fig. 1-9(b), only four registers have to be reset, which is not a problem, but in real MMUs, many more registers have to be reloaded, either explicitly or dynamically, as needed. Either way, it takes time. The moral of the story is that switching from one program to another, called a **context switch**, is an expensive business.

1.4.3 I/O Devices

Memory is not the only resource that the operating system must manage. I/O devices also interact heavily with the operating system. As we saw in Fig. 1-5, I/O devices generally consist of two parts: a controller and the device itself. The controller is a chip or a set of chips on a plug-in board that physically controls the device. It accepts commands from the operating system, for example, to read data from the device, and carries them out.

In many cases, the actual control of the device is very complicated and detailed, so it is the job of the controller to present a simpler interface to the operating system. For example, a disk controller might accept a command to read sector 11,206 from disk 2. The controller then has to convert this linear sector number to a cylinder, sector, and head. This conversion may be complicated by the fact that outer cylinders have more sectors than inner ones and that some bad sectors have been remapped onto other ones. Then the controller has to determine which cylinder the disk arm is on and give it a sequence of pulses to move in or out the requisite number of cylinders. It has to wait until the proper sector has rotated under the head and then start reading and storing the bits as they come off the drive, removing the preamble and computing the checksum. Finally, it has to assemble the incoming bits into words and store them in memory. To do all this work, controllers often contain small embedded computers that are programmed to do their work.

The other piece is the actual device itself. Devices have fairly simple interfaces, both because they cannot do much and to make them standard. The latter is needed so that any IDE disk controller can handle any IDE disk, for example. **IDE** stands for **Integrated Drive Electronics** and is the standard type of disk on Pentiums and some other computers. Since the actual device interface is hidden

Spotify USA Inc. Exhibit 1009 Page 48

behind the controller, all that the operating system sees is the interface to the controller, which may be quite different from the interface to the device.

Because each type of controller is different, different software is needed to control each one. The software that talks to a controller, giving it commands and accepting responses, is called a **device driver**. Each controller manufacturer has to supply a driver for each operating system it supports. Thus a scanner may come with drivers for Windows 98, Windows 2000, and UNIX, for example.

To be used, the driver has to be put into the operating system so it can run in kernel mode. Theoretically, drivers can run outside the kernel, but few current systems support this possibility because it requires the ability to allow a user-space driver to be able to access the device in a controlled way, a feature rarely supported. There are three ways the driver can be put into the kernel. The first way is to relink the kernel with the new driver and then reboot the system. Many UNIX systems work like this. The second way is to make an entry in an operating system file telling it that it needs the driver and then reboot the system. At boot time, the operating system goes and finds the drivers it needs and loads them. Windows works this way. The third way is for the operating system to be able to accept new drivers while running and install them on-the-fly without the need to reboot. This way used to be rare but is becoming much more common now. Hot pluggable devices, such as USB and IEEE 1394 devices (discussed below) always need dynamically loaded drivers.

Every controller has a small number of registers that are used to communicate with it. For example, a minimal disk controller might have registers for specifying the disk address, memory address, sector count, and direction (read or write). To activate the controller, the driver gets a command from the operating system, then translates it into the appropriate values to write into the device registers.

On some computers, the device registers are mapped into the operating system's address space, so they can be read and written like ordinary memory words. On such computers, no special I/O instructions are needed and user programs can be kept away from the hardware by not putting these memory addresses within their reach (e.g., by using base and limit registers). On other computers, the device registers are put in a special I/O port space, with each register having a port address. On these machines, special IN and OUT instructions are available in kernel mode to allow drivers to read and write the registers. The former scheme eliminates the need for special I/O instructions but uses up some of the address space. The latter uses no address space but requires special instructions. Both systems are widely used.

Input and output can be done in three different ways. In the simplest method, a user program issues a system call, which the kernel then translates into a procedure call to the appropriate driver. The driver then starts the I/O and sits in a tight loop continuously polling the device to see if it is done (usually there is some bit that indicates that the device is still busy). When the I/O has completed, the driver puts the data where they are needed (if any), and returns. The operating

Spotify USA Inc. Exhibit 1009 Page 49

system then returns control to the caller. This method is called **busy waiting** and has the disadvantage of tying up the CPU polling the device until it is finished.

The second method is for the driver to start the device and ask it to give an interrupt when it is finished. At that point the driver returns. The operating system then blocks the caller if need be and looks for other work to do. When the controller detects the end of the transfer, it generates an **interrupt** to signal completion.

Interrupts are very important in operating systems, so let us examine the idea more closely. In Fig. 1-10(a) we see a three-step process for I/O. In step 1, the driver tells the controller what to do by writing into its device registers. The controller then starts the device. When the controller has finished reading or writing the number of bytes it has been told to transfer, it signals the interrupt controller chip using certain bus lines in step 2. If the interrupt controller is prepared to accept the interrupt (which it may not be if it is busy with a higher priority one), it asserts a pin on the CPU chip informing it, in step 3. In step 4, the interrupt controller puts the number of the device on the bus so the CPU can read it and know which device has just finished (many devices may be running at the same time).

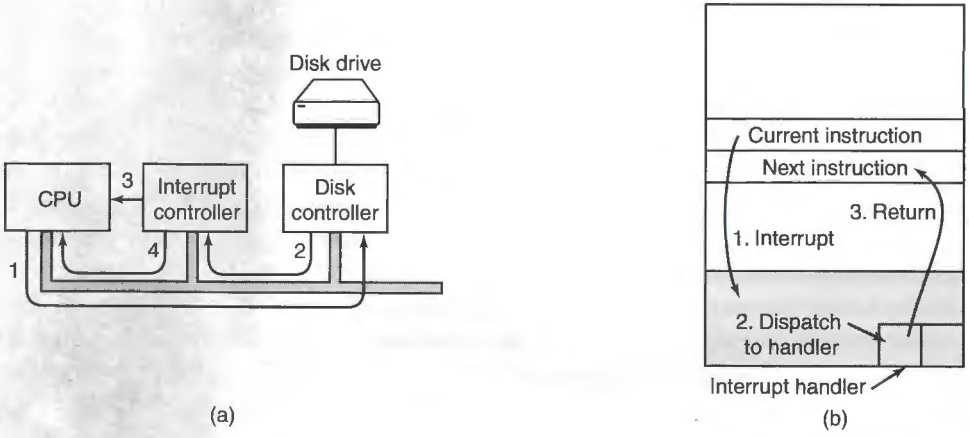


Figure 1-10. (a) The steps in starting an I/O device and getting an interrupt. (b) Interrupt processing involves taking the interrupt, running the interrupt handler, and returning to the user program.

Once the CPU has decided to take the interrupt, the program counter and PSW are typically then pushed onto the current stack and the CPU switched into kernel mode. The device number may be used as an index into part of memory to find the address of the interrupt handler for this device. This part of memory is called the **interrupt vector**. Once the interrupt handler (part of the driver for the interrupting device) has started, it removes the stacked program counter and PSW and saves them, then queries the device to learn its status. When the handler is all finished, it returns to the previously-running user program to the first instruction that was not yet executed. These steps are shown in Fig. 1-10(b).

Spotify USA Inc. Exhibit 1009 Page 50

The third method for doing I/O makes use of a special **DMA** (**Direct Memory Access**) chip that can control the flow of bits between memory and some controller without constant CPU intervention. The CPU sets up the DMA chip, telling it how many bytes to transfer, the device and memory addresses involved, and the direction, and lets it go. When the DMA chip is done, it causes an interrupt, which is handled as described above. DMA and I/O hardware in general will be discussed in more detail in Chap. 5.

Interrupts can often happen at highly inconvenient moments, for example, while another interrupt handler is running. For this reason, the CPU has a way to disable interrupts and then reenable them later. While interrupts are disabled, any devices that finish continue to assert their interrupt signals, but the CPU is not interrupted until interrupts are enabled again. If multiple devices finish while interrupts are disabled, the interrupt controller decides which one to let through first, usually based on static priorities assigned to each device. The highest priority device wins.

1.4.4 Buses

The organization of Fig. 1-5 was used on minicomputers for years and also on the original IBM PC. However, as processors and memories got faster, the ability of a single bus (and certainly the IBM PC bus) to handle all the traffic was strained to the breaking point. Something had to give. As a result, additional buses were added, both for faster I/O devices and for CPU to memory traffic. As a consequence of this evolution, a large Pentium system currently looks something like Fig. 1-11.

This system has eight buses (cache, local, memory, PCI, SCSI, USB, IDE, and ISA), each with a different transfer rate and function. The operating system must be aware of all of them for configuration and management. The two main buses are the original IBM PC **ISA** (**Industry Standard Architecture**) bus and its successor, the **PCI** (**Peripheral Component Interconnect**) bus. The ISA bus, which was originally the IBM PC/AT bus, runs at 8.33 MHz and can transfer 2 bytes at once, for a maximum speed of 16.67 MB/sec. It is included for backward compatibility with old and slow I/O cards. The PCI bus was invented by Intel as a successor to the ISA bus. It can run at 66 MHz and transfer 8 bytes at a time, for a data rate of 528 MB/sec. Most high-speed I/O devices use the PCI bus now. Even some non-Intel computers use the PCI bus due to the large number of I/O cards available for it.

In this configuration, the CPU talks to the PCI bridge chip over the local bus, and the PCI bridge chip talks to the memory over a dedicated memory bus, often running at 100 MHz. Pentium systems have a level-1 cache on chip and a much larger level-2 cache off chip, connected to the CPU by the cache bus.

In addition, this system contains three specialized buses: IDE, USB, and SCSI. The IDE bus is for attaching peripheral devices such as disks and CD-

Spotify USA Inc. Exhibit 1009 Page 51

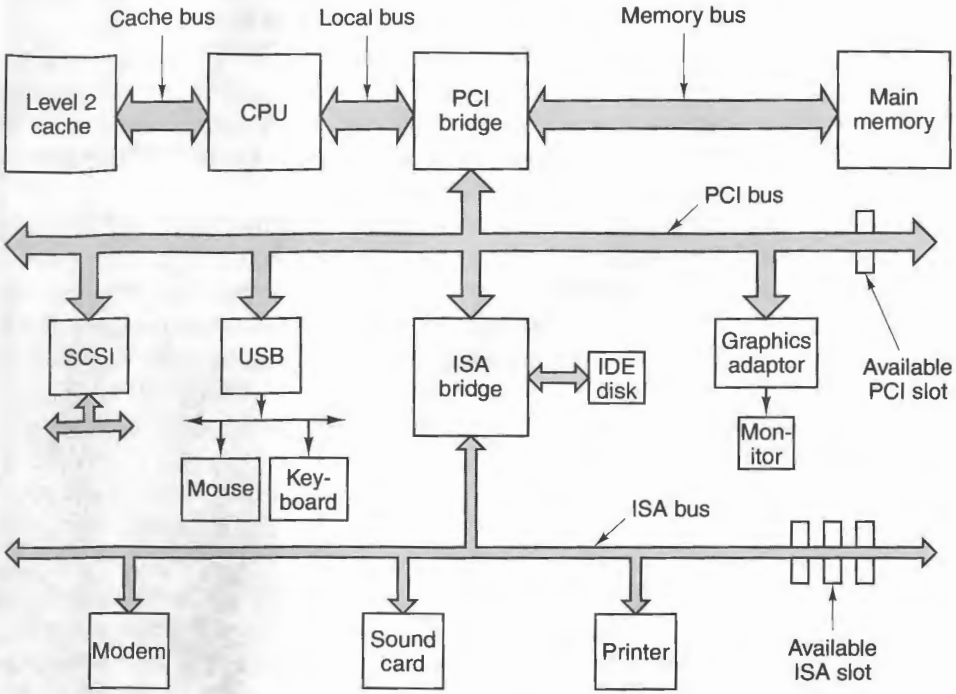


Figure 1-11. The structure of a large Pentium system

ROMs to the system. The IDE bus is an outgrowth of the disk controller interface on the PC/AT and is now standard on nearly all Pentium-based systems for the hard disk and often the CD-ROM.

The **USB (Universal Serial Bus)** was invented to attach all the slow I/O devices, such as the keyboard and mouse, to the computer. It uses a small four-wire connector, two of which supply electrical power to the USB devices. USB is a centralized bus in which a root device polls the I/O devices every 1 msec to see if they have any traffic. It can handle an aggregate load of 1.5 MB/sec. All the USB devices share a single USB device driver, making it unnecessary to install a new driver for each new USB device. Consequently, USB devices can be added to the computer without the need to reboot.

The **SCSI (Small Computer System Interface)** bus is a high-performance bus intended for fast disks, scanners, and other devices needing considerable bandwidth. It can run at up to 160 MB/sec. It has been present on Macintosh systems since they were invented and is also popular on UNIX and some Intel-based systems.

Yet another bus (not shown in Fig. 1-11) is **IEEE 1394**. Sometimes it is called FireWire, although strictly speaking, FireWire is the name Apple uses for its implementation of 1394. Like USB, IEEE 1394 is bit serial but is designed for

Spotify USA Inc. Exhibit 1009 Page 52

packet transfers at speeds up to 50 MB/sec, making it useful for connecting digital camcorders and similar multimedia devices to a computer. Unlike USB, IEEE 1394 does not have a central controller. SCSI and IEEE 1394 face competition from a faster version of USB being developed.

To work in an environment such as that of Fig. 1-11, the operating system has to know what is out there and configure it. This requirement led Intel and Microsoft to design a PC system called **plug and play**, based on a similar concept first implemented in the Apple Macintosh. Before plug and play, each I/O card had a fixed interrupt request level and fixed addresses for its I/O registers. For example, the keyboard was interrupt 1 and used I/O addresses 0x60 to 0x64, the floppy disk controller was interrupt 6 and used I/O addresses 0x3F0 to 0x3F7, and the printer was interrupt 7 and used I/O addresses 0x378 to 0x37A, and so on.

So far, so good. The trouble came when the user bought a sound card and a modem card and both happened to use, say, interrupt 4. They would conflict and would not work together. The solution was to include DIP switches or jumpers on every I/O card and instruct the user to please set them to select an interrupt level and I/O device addresses that did not conflict with any others in the user's system. Teenagers who devoted their lives to the intricacies of the PC hardware could sometimes do this without making errors. Unfortunately, nobody else could, leading to chaos.

What plug and play does is have the system automatically collect information about the I/O devices, centrally assign interrupt levels and I/O addresses, and then tell each card what its numbers are. Very briefly, that works as follows on the Pentium. Every Pentium contains a parentboard (formerly called a motherboard before political correctness hit the computer industry). On the parentboard is a program called the system **BIOS** (**Basic Input Output System**) The BIOS contains low-level I/O software, including procedures to read the keyboard, write to the screen, and do disk I/O, among other things. Nowadays, it is held in a flash RAM, which is nonvolatile but which can be updated by the operating system when bugs are found in the BIOS.

When the computer is booted, the BIOS is started. It first checks to see how much RAM is installed and whether the keyboard and other basic devices are installed and responding correctly. It starts out by scanning the ISA and PCI buses to detect all the devices attached to them. Some of these devices are typically **legacy** (i.e., designed before plug and play was invented) and have fixed interrupt levels and I/O addresses (possibly set by switches or jumpers on the I/O card, but not modifiable by the operating system). These devices are recorded. The plug and play devices are also recorded. If the devices present are different from when the system was last booted, the new devices are configured.

The BIOS then determines the boot device by trying a list of devices stored in the CMOS memory. The user can change this list by entering a BIOS configuration program just after booting. Typically, an attempt is made to boot from the floppy disk. If that fails the CD-ROM is tried. If neither a floppy nor a CD-ROM

Spotify USA Inc. Exhibit 1009 Page 53

is present, the system is booted from the hard disk. The first sector from the boot device is read into memory and executed. This sector contains a program that normally examines the partition table at the end of the boot sector to determine which partition is active. Then a secondary boot loader is read in from that partition. This loader reads in the operating system from the active partition and starts it.

The operating system then queries the BIOS to get the configuration information. For each device, it checks to see if it has the device driver. If not, it asks the user to insert a floppy disk or CD-ROM containing the driver (supplied by the device's manufacturer). Once it has all the device drivers, the operating system loads them into the kernel. Then it initializes its tables, creates whatever background processes are needed, and starts up a login program or GUI on each terminal. At least, this is the way it is supposed to work. In real life, plug and play is frequently so unreliable that many people call it plug and pray.

1.5 OPERATING SYSTEM CONCEPTS

All operating systems have certain basic concepts such as processes, memory, and files that are central to understanding them. In the following sections, we will look at some of these basic concepts ever so briefly, as an introduction. We will come back to each of them in great detail later in this book. To illustrate these concepts we will use examples from time to time, generally drawn from UNIX. Similar examples typically exist in other systems as well, however.

1.5.1 Processes

A key concept in all operating systems is the **process**. A process is basically a program in execution. Associated with each process is its **address space**, a list of memory locations from some minimum (usually 0) to some maximum, which the process can read and write. The address space contains the executable program, the program's data, and its stack. Also associated with each process is some set of registers, including the program counter, stack pointer, and other hardware registers, and all the other information needed to run the program.

We will come back to the process concept in much more detail in Chap. 2, but for the time being, the easiest way to get a good intuitive feel for a process is to think about timesharing systems. Periodically, the operating system decides to stop running one process and start running another, for example, because the first one has had more than its share of CPU time in the past second.

When a process is suspended temporarily like this, it must later be restarted in exactly the same state it had when it was stopped. This means that all information about the process must be explicitly saved somewhere during the suspension. For example, the process may have several files open for reading at once. Associated

Spotify USA Inc. Exhibit 1009 Page 54

with each of these files is a pointer giving the current position (i.e., the number of the byte or record to be read next). When a process is temporarily suspended, all these pointers must be saved so that a read call executed after the process is restarted will read the proper data. In many operating systems, all the information about each process, other than the contents of its own address space, is stored in an operating system table called the **process table**, which is an array (or linked list) of structures, one for each process currently in existence.

Thus, a (suspended) process consists of its address space, usually called the **core image** (in honor of the magnetic core memories used in days of yore), and its process table entry, which contains its registers, among other things.

The key process management system calls are those dealing with the creation and termination of processes. Consider a typical example. A process called the **command interpreter** or **shell** reads commands from a terminal. The user has just typed a command requesting that a program be compiled. The shell must now create a new process that will run the compiler. When that process has finished the compilation, it executes a system call to terminate itself.

If a process can create one or more other processes (referred to as **child processes**) and these processes in turn can create child processes, we quickly arrive at the process tree structure of Fig. 1-12. Related processes that are cooperating to get some job done often need to communicate with one another and synchronize their activities. This communication is called **interprocess communication**, and will be addressed in detail in Chap. 2.

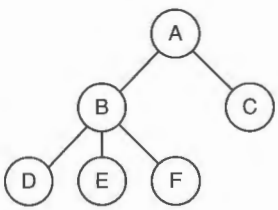


Figure 1-12. A process tree. Process *A* created two child processes, *B* and *C*. Process *B* created three child processes, *D*, *E*, and *F*.

Other process system calls are available to request more memory (or release unused memory), wait for a child process to terminate, and overlay its program with a different one.

Occasionally, there is a need to convey information to a running process that is not sitting around waiting for this information. For example, a process that is communicating with another process on a different computer does so by sending messages to the remote process over a computer network. To guard against the possibility that a message or its reply is lost, the sender may request that its own operating system notify it after a specified number of seconds, so that it can retransmit the message if no acknowledgement has been received yet. After setting this timer, the program may continue doing other work.

Spotify USA Inc. Exhibit 1009 Page 55

When the specified number of seconds has elapsed, the operating system sends an **alarm signal** to the process. The signal causes the process to temporarily suspend whatever it was doing, save its registers on the stack, and start running a special signal handling procedure, for example, to retransmit a presumably lost message. When the signal handler is done, the running process is restarted in the state it was in just before the signal. Signals are the software analog of hardware interrupts and can be generated by a variety of causes in addition to timers expiring. Many traps detected by hardware, such as executing an illegal instruction or using an invalid address, are also converted into signals to the guilty process.

Each person authorized to use a system is assigned a **UID** (User IDentification) by the system administrator. Every process started has the UID of the person who started it. A child process has the same UID as its parent. Users can be members of groups, each of which has a **GID (Group IDentification)**.

One UID, called the **superuser** (in UNIX), has special power and may violate many of the protection rules. In large installations, only the system administrator knows the password needed to become superuser, but many of the ordinary users (especially students) devote considerable effort to trying to find flaws in the system that allow them to become superuser without the password.

We will study processes, interprocess communication, and related issues in Chap. 2.

1.5.2 Deadlocks

When two or more processes are interacting, they can sometimes get themselves into a stalemate situation they cannot get out of. Such a situation is called a deadlock.

Deadlocks can best be introduced with a real-world example everyone is familiar with, deadlock in traffic. Consider the situation of Fig. 1-13(a). Here four buses are approaching an intersection. Behind each one are more buses (not shown). With a little bit of bad luck, the first four could all arrive at the intersection simultaneously, leading to the situation of Fig. 1-13(b), in which they are deadlocked because none of them can go forward. Each one is blocking one of the others. They cannot go backward due to other buses behind them. There is no easy way out.

Processes in a computer can experience an analogous situation in which they cannot make any progress. For example, imagine a computer with a tape drive and CD-recorder. Now imagine that two processes each need to produce a CD-ROM from data on a tape. Process 1 requests and is granted the tape drive. Next process 2 requests and is granted the CD-recorder. Then process 1 requests the CD-recorder and is suspended until process 2 returns it. Finally, process 2 requests the tape drive and is also suspended because process 1 already has it. Here

Spotify USA Inc. Exhibit 1009 Page 56



Figure 1-13. (a) A potential deadlock. (b) An actual deadlock.

we have a deadlock from which there is no escape. We will study deadlocks and what can be done about them in detail in Chap. 3.

1.5.3 Memory Management

Every computer has some main memory that it uses to hold executing programs. In a very simple operating system, only one program at a time is in memory. To run a second program, the first one has to be removed and the second one placed in memory.

More sophisticated operating systems allow multiple programs to be in memory at the same time. To keep them from interfering with one another (and with the operating system), some kind of protection mechanism is needed. While this mechanism has to be in the hardware, it is controlled by the operating system.

The above viewpoint is concerned with managing and protecting the computer's main memory. A different, but equally important memory-related issue, is managing the address space of the processes. Normally, each process has some set of addresses it can use, typically running from 0 up to some maximum. In the simplest case, the maximum amount of address space a process has is less than the main memory. In this way, a process can fill up its address space and there will be enough room in main memory to hold it all.

However, on many computers addresses are 32 or 64 bits, giving an address space of 2^{32} or 2^{64} bytes, respectively. What happens if a process has more address space than the computer has main memory and the process wants to use it all? In the first computers, such a process was just out of luck. Nowadays, a technique called virtual memory exists, in which the operating system keeps part of the address space in main memory and part on disk and shuttles pieces back and forth between them as needed. This important operating system function, and other memory management-related functions will be covered in Chap. 4.

Spotify USA Inc. Exhibit 1009 Page 57

1.5.4 Input/Output

All computers have physical devices for acquiring input and producing output. After all, what good would a computer be if the users could not tell it what to do and could not get the results after it did the work requested. Many kinds of input and output devices exist, including keyboards, monitors, printers, and so on. It is up to the operating system to manage these devices.

Consequently, every operating system has an I/O subsystem for managing its I/O devices. Some of the I/O software is device independent, that is, applies to many or all I/O devices equally well. Other parts of it, such as device drivers, are specific to particular I/O devices. In Chap. 5 we will have a look at I/O software.

1.5.5 Files

Another key concept supported by virtually all operating systems is the file system. As noted before, a major function of the operating system is to hide the peculiarities of the disks and other I/O devices and present the programmer with a nice, clean abstract model of device-independent files. System calls are obviously needed to create files, remove files, read files, and write files. Before a file can be read, it must be located on the disk and opened, and after it has been read it should be closed, so calls are provided to do these things.

To provide a place to keep files, most operating systems have the concept of a **directory** as a way of grouping files together. A student, for example, might have one directory for each course he is taking (for the programs needed for that course), another directory for his electronic mail, and still another directory for his World Wide Web home page. System calls are then needed to create and remove directories. Calls are also provided to put an existing file in a directory, and to remove a file from a directory. Directory entries may be either files or other directories. This model also gives rise to a hierarchy—the file system—as shown in Fig. 1-14.

The process and file hierarchies both are organized as trees, but the similarity stops there. Process hierarchies usually are not very deep (more than three levels is unusual), whereas file hierarchies are commonly four, five, or even more levels deep. Process hierarchies are typically short-lived, generally a few minutes at most, whereas the directory hierarchy may exist for years. Ownership and protection also differ for processes and files. Typically, only a parent process may control or even access a child process, but mechanisms nearly always exist to allow files and directories to be read by a wider group than just the owner.

Every file within the directory hierarchy can be specified by giving its **path name** from the top of the directory hierarchy, the **root directory**. Such absolute path names consist of the list of directories that must be traversed from the root directory to get to the file, with slashes separating the components. In Fig. 1-14,

Spotify USA Inc. Exhibit 1009 Page 58

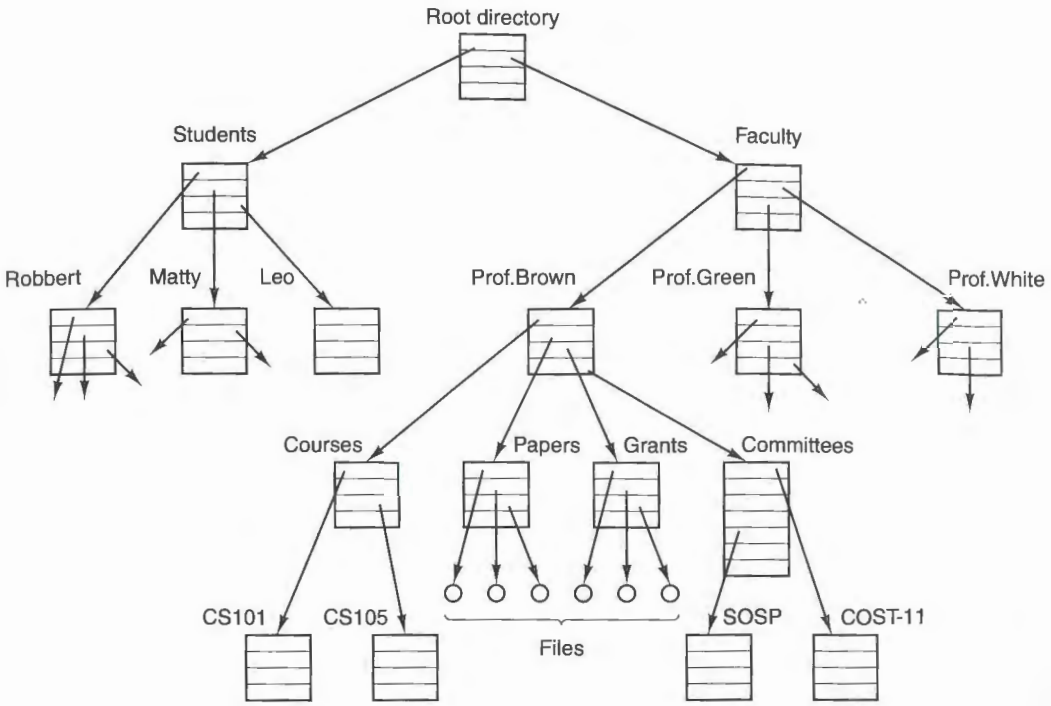


Figure 1-14. A file system for a university department.

the path for file *CS101* is */Faculty/Prof.Brown/Courses/CS101*. The leading slash indicates that the path is absolute, that is, starting at the root directory. As an aside, in MS-DOS and Windows, the backslash (\) character is used as the separator instead of the slash (/) character, so the file path given above would be written as *\Faculty\Prof.Brown\Courses\CS101*. Throughout this book we will generally use the UNIX convention for paths.

At every instant, each process has a current **working directory**, in which path names not beginning with a slash are looked for. As an example, in Fig. 1-14, if */Faculty/Prof.Brown* were the working directory, then use of the path name *Courses/CS101* would yield the same file as the absolute path name given above. Processes can change their working directory by issuing a system call specifying the new working directory.

Before a file can be read or written, it must be opened, at which time the permissions are checked. If the access is permitted, the system returns a small integer called a **file descriptor** to use in subsequent operations. If the access is prohibited, an error code is returned.

Another important concept in UNIX is the mounted file system. Nearly all personal computers have one or more floppy disk drives into which floppy disks can be inserted and removed. To provide an elegant way to deal with removable

Spotify USA Inc. Exhibit 1009 Page 59

media (including CD-ROMs), UNIX allows the file system on a floppy disk to be attached to the main tree. Consider the situation of Fig. 1-15(a). Before the mount call, the **root file system**, on the hard disk, and a second file system, on a floppy disk, are separate and unrelated.

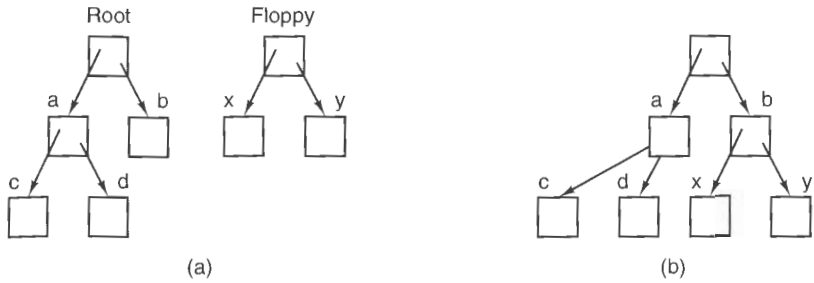


Figure 1-15. (a) Before mounting, the files on drive 0 are not accessible. (b) After mounting, they are part of the file hierarchy.

However, the file system on the floppy cannot be used, because there is no way to specify path names on it. UNIX does not allow path names to be prefixed by a drive name or number; that would be precisely the kind of device dependence that operating systems ought to eliminate. Instead, the mount system call allows the file system on the floppy to be attached to the root file system wherever the program wants it to be. In Fig. 1-15(b) the file system on the floppy has been mounted on directory *b*, thus allowing access to files */b/x* and */b/y*. If directory *b* had contained any files they would not be accessible while the floppy was mounted, since */b* would refer to the root directory of the floppy. (Not being able to access these files is not as serious as it at first seems: file systems are nearly always mounted on empty directories.) If a system contains multiple hard disks, they can all be mounted into a single tree as well.

Another important concept in UNIX is the **special file**. Special files are provided in order to make I/O devices look like files. That way, they can be read and written using the same system calls as are used for reading and writing files. Two kinds of special files exist: **block special files** and **character special files**. Block special files are used to model devices that consist of a collection of randomly addressable blocks, such as disks. By opening a block special file and reading, say, block 4, a program can directly access the fourth block on the device, without regard to the structure of the file system contained on it. Similarly, character special files are used to model printers, modems, and other devices that accept or output a character stream. By convention, the special files are kept in the */dev* directory. For example, */dev/lp* might be the line printer.

The last feature we will discuss in this overview is one that relates to both processes and files: pipes. A **pipe** is a sort of pseudofile that can be used to connect two processes, as shown in Fig. 1-16. If processes *A* and *B* wish to talk using

Spotify USA Inc. Exhibit 1009 Page 60

a pipe, they must set it up in advance. When process *A* wants to send data to process *B*, it writes on the pipe as though it were an output file. Process *B* can read the data by reading from the pipe as though it were an input file. Thus, communication between processes in UNIX looks very much like ordinary file reads and writes. Stronger yet, the only way a process can discover that the output file it is writing on is not really a file, but a pipe, is by making a special system call. File systems are very important. We will have much more to say about them in Chap. 6 and also in Chaps. 10 and 11.

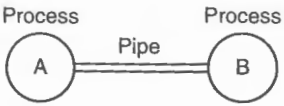


Figure 1-16. Two processes connected by a pipe.

1.5.6 Security

Computers contain large amounts of information that users often want to keep confidential. This information may include electronic mail, business plans, tax returns, and much more. It is up to the operating system to manage the system security so that files, for example, are only accessible to authorized users.

As a simple example, just to get an idea of how security can work, consider UNIX. Files in UNIX are protected by assigning each one a 9-bit binary protection code. The protection code consists of three 3-bit fields, one for the owner, one for other members of the owner's group (users are divided into groups by the system administrator), and one for everyone else. Each field has a bit for read access, a bit for write access, and a bit for execute access. These 3 bits are known as the **rwx bits**. For example, the protection code *rwxr-x--x* means that the owner can read, write, or execute the file, other group members can read or execute (but not write) the file, and everyone else can execute (but not read or write) the file. For a directory, *x* indicates search permission. A dash means that the corresponding permission is absent.

In addition to file protection, there are many other security issues. Protecting the system from unwanted intruders, both human and nonhuman (e.g., viruses) is one of them. We will look at various security issues in Chap. 9.

1.5.7 The Shell

The operating system is the code that carries out the system calls. Editors, compilers, assemblers, linkers, and command interpreters definitely are not part of the operating system, even though they are important and useful. At the risk of confusing things somewhat, in this section we will look briefly at the UNIX com-

Spotify USA Inc. Exhibit 1009 Page 61

mand interpreter, called the **shell**. Although it is not part of the operating system, it makes heavy use of many operating system features and thus serves as a good example of how the system calls can be used. It is also the primary interface between a user sitting at his terminal and the operating system, unless the user is using a graphical user interface. Many shells exist, including *sh*, *csh*, *ksh*, and *bash*. All of them support the functionality described below, which derives from the original shell (*sh*).

When any user logs in, a shell is started up. The shell has the terminal as standard input and standard output. It starts out by typing the **prompt**, a character such as a dollar sign, which tells the user that the shell is waiting to accept a command. If the user now types

```
date
```

for example, the shell creates a child process and runs the *date* program as the child. While the child process is running, the shell waits for it to terminate. When the child finishes, the shell types the prompt again and tries to read the next input line.

The user can specify that standard output be redirected to a file, for example,

```
date >file
```

Similarly, standard input can be redirected, as in

```
sort <file1 >file2
```

which invokes the sort program with input taken from *file1* and output sent to *file2*.

The output of one program can be used as the input for another program by connecting them with a pipe. Thus

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

invokes the *cat* program to con*cat*enate three files and send the output to *sort* to arrange all the lines in alphabetical order. The output of *sort* is redirected to the file */dev/lp*, typically the printer.

If a user puts an ampersand after a command, the shell does not wait for it to complete. Instead it just gives a prompt immediately. Consequently,

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

starts up the sort as a background job, allowing the user to continue working normally while the sort is going on. The shell has a number of other interesting features, which we do not have space to discuss here. Most books on UNIX discuss the shell at some length (e.g., Kernighan and Pike, 1984; Kochan and Wood, 1990; Medinets, 1999; Newham and Rosenblatt, 1998; and Robbins, 1999).

Spotify USA Inc. Exhibit 1009 Page 62

1.5.8 Recycling of Concepts

Computer science, like many fields, is largely technology driven. The reason the ancient Romans lacked cars is not that they liked walking so much. It is because they did not know how to build cars. Personal computers exist *not* because millions of people had some long pent-up desire to own a computer, but because it is now possible to manufacture them cheaply. We often forget how much technology affects our view of systems and it is worth reflecting on this point from time to time.

In particular, it frequently happens that a change in technology renders some idea obsolete and it quickly vanishes. However, another change in technology could revive it again. This is especially true when the change has to do with the relative performance of different parts of the system. For example, when CPUs became much faster than memories, caches became important to speed up the "slow" memory. If new memory technology some day makes memories much faster than CPUs, caches will vanish. And if a new CPU technology makes them faster than memories again, caches will reappear. In biology, extinction is forever, but in computer science, it is sometimes only for a few years.

As a consequence of this impermanence, in this book we will from time to time look at "obsolete" concepts, that is, ideas that are not optimal with current technology. However, changes in the technology may bring back some of the so-called "obsolete concepts." For this reason, it is important to understand why a concept is obsolete and what changes in the environment might bring it back again.

To make this point clearer, let us consider a few examples. Early computers had hardwired instruction sets. The instructions were executed directly by hardware and could not be changed. Then came microprogramming, in which an underlying interpreter carried out the instructions in software. Hardwired execution became obsolete. Then RISC computers were invented, and microprogramming (i.e., interpreted execution) became obsolete because direct execution was faster. Now we are seeing the resurgence of interpretation in the form of Java applets that are sent over the Internet and interpreted upon arrival. Execution speed is not always crucial because network delays are so great that they tend to dominate. But that could change, too, some day.

Early operating systems allocated files on the disk by just placing them in contiguous sectors, one after another. Although this scheme was easy to implement, it was not flexible because when a file grew, there was not enough room to store it any more. Thus the concept of contiguously allocated files was discarded as obsolete. Until CD-ROMs came around. There the problem of growing files did not exist. All of a sudden, the simplicity of contiguous file allocation was seen as a great idea and CD-ROM file systems are now based on it.

As our final idea, consider dynamic linking. The MULTICS system was designed to run day and night without ever stopping. To fix bugs in software, it

was necessary to have a way to replace library procedures while they were being used. The concept of dynamic linking was invented for this purpose. After MULTICS died, the concept was forgotten for a while. However, it was rediscovered when modern operating systems needed a way to allow many programs to share the same library procedures without having their own private copies (because graphics libraries had grown so large). Most systems now support some form of dynamic linking once again. The list goes on, but these examples should make the point: an idea that is obsolete today may be the star of the party tomorrow.

Technology is not the only factor that drives systems and software. Economics plays a big role too. In the 1960s and 1970s, most terminals were mechanical printing terminals or 25×80 character-oriented CRTs rather than bitmap graphics terminals. This choice was not a question of technology. Bit-map graphics terminals were in use before 1960. It is just that they cost many tens of thousands of dollars each. Only when the price came down enormously could people (other than the military) think of dedicating one terminal to an individual user.

1.6 SYSTEM CALLS

The interface between the operating system and the user programs is defined by the set of system calls that the operating system provides. To really understand what operating systems do, we must examine this interface closely. The system calls available in the interface vary from operating system to operating system (although the underlying concepts tend to be similar).

We are thus forced to make a choice between (1) vague generalities ("operating systems have system calls for reading files") and (2) some specific system ("UNIX has a read system call with three parameters: one to specify the file, one to tell where the data are to be put, and one to tell how many bytes to read").

We have chosen the latter approach. It's more work that way, but it gives more insight into what operating systems really do. Although this discussion specifically refers to POSIX (International Standard 9945-1), hence also to UNIX, System V, BSD, Linux, MINIX, etc., most other modern operating systems have system calls that perform the same functions, even if the details differ. Since the actual mechanics of issuing a system call are highly machine dependent and often must be expressed in assembly code, a procedure library is provided to make it possible to make system calls from C programs and often from other languages as well.

It is useful to keep the following in mind. Any single-CPU computer can execute only one instruction at a time. If a process is running a user program in user mode and needs a system service, such as reading data from a file, it has to execute a trap or system call instruction to transfer control to the operating system. The operating system then figures out what the calling process wants by inspecting the parameters. Then it carries out the system call and returns control to the

Spotify USA Inc. Exhibit 1009 Page 64

instruction following the system call. In a sense, making a system call is like making a special kind of procedure call, only system calls enter the kernel and procedure calls do not.

To make the system call mechanism clearer, let us take a quick look at the read system call. As mentioned above, it has three parameters: the first one specifying the file, the second one pointing to the buffer, and the third one giving the number of bytes to read. Like nearly all system calls, it is invoked from C programs by calling a library procedure with the same name as the system call: *read*. A call from a C program might look like this:

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

The system call (and the library procedure) return the number of bytes actually read in *count*. This value is normally the same as *nbytes*, but may be smaller, if, for example, end-of-file is encountered while reading.

If the system call cannot be carried out, either due to an invalid parameter or a disk error, *count* is set to −1, and the error number is put in a global variable, *errno*. Programs should always check the results of a system call to see if an error occurred.

System calls are performed in a series of steps. To make this concept clearer, let us examine the read call discussed above. In preparation for calling the *read* library procedure, which actually makes the read system call, the calling program first pushes the parameters onto the stack, as shown in steps 1-3 in Fig. 1-17. C and C++ compilers push the parameters onto the stack in reverse order for historical reasons (having to do with making the first parameter to *printf*, the format string, appear on top of the stack). The first and third parameters are called by value, but the second parameter is passed by reference, meaning that the address of the buffer (indicated by &) is passed, not the contents of the buffer. Then comes the actual call to the library procedure (step 4). This instruction is the normal procedure call instruction used to call all procedures.

The library procedure, possibly written in assembly language, typically puts the system call number in a place where the operating system expects it, such as a register (step 5). Then it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel (step 6). The kernel code that starts examines the system call number and then dispatches to the correct system call handler, usually via a table of pointers to system call handlers indexed on system call number (step 7). At that point the system call handler runs (step 8). Once the system call handler has completed its work, control may be returned to the user-space library procedure at the instruction following the TRAP instruction (step 9). This procedure then returns to the user program in the usual way procedure calls return (step 10).

To finish the job, the user program has to clean up the stack, as it does after any procedure call (step 11). Assuming the stack grows downward, as it often does, the compiled code increments the stack pointer exactly enough to remove

Spotify USA Inc. Exhibit 1009 Page 65

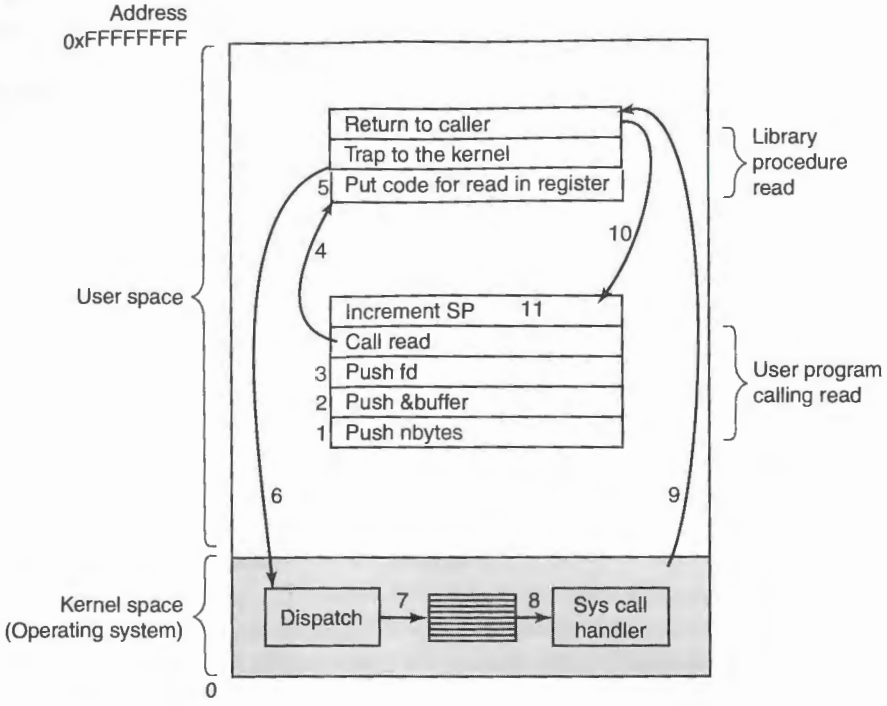


Figure 1-17. The 11 steps in making the system call read(fd, buffer, nbytes).

the parameters pushed before the call to *read*. The program is now free to do whatever it wants to do next.

In step 9 above, we said "may be returned to the user-space library procedure ..." for good reason. The system call may block the caller, preventing it from continuing. For example, if it is trying to read from the keyboard and nothing has been typed yet, the caller has to be blocked. In this case, the operating system will look around to see if some other process can be run next. Later, when the desired input is available, this process will get the attention of the system and steps 9–11 will occur.

In the following sections, we will examine some of the most heavily used POSIX system calls, or more specifically, the library procedures that make those system calls. POSIX has about 100 procedure calls. Some of the most important ones are listed in Fig. 1-18, grouped for convenience in four categories. In the text we will briefly examine each call to see what it does. To a large extent, the services offered by these calls determine most of what the operating system has to do, since the resource management on personal computers is minimal (at least compared to big machines with multiple users). The services include things like creating and terminating processes, creating, deleting, reading, and writing files, managing directories, and performing input and output.

Spotify USA Inc. Exhibit 1009 Page 66

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Figure 1-18. Some of the major POSIX system calls. The return code *s* is −1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time. The parameters are explained in the text.

As an aside, it is worth pointing out that the mapping of POSIX procedure calls onto system calls is not one-to-one. The POSIX standard specifies a number of procedures that a conformant system must supply, but it does not specify

Spotify USA Inc. Exhibit 1009 Page 67

whether they are system calls, library calls, or something else. If a procedure can be carried out without invoking a system call (i.e., without trapping to the kernel), it will usually be done in user space for reasons of performance. However, most of the POSIX procedures do invoke system calls, usually with one procedure mapping directly onto one system call. In a few cases, especially where several required procedures are only minor variations of one another, one system call handles more than one library call.

1.6.1 System Calls for Process Management

The first group of calls in Fig. 1-18 deals with process management. Fork is a good place to start the discussion. Fork is the only way to create a new process in UNIX. It creates an exact duplicate of the original process, including all the file descriptors, registers—everything. After the fork, the original process and the copy (the parent and child) go their separate ways. All the variables have identical values at the time of the fork, but since the parent's data are copied to create the child, subsequent changes in one of them do not affect the other one. (The program text, which is unchangeable, is shared between parent and child.) The fork call returns a value, which is zero in the child and equal to the child's process identifier or **PID** in the parent. Using the returned PID, the two processes can see which one is the parent process and which one is the child process.

In most cases, after a fork, the child will need to execute different code from the parent. Consider the case of the shell. It reads a command from the terminal, forks off a child process, waits for the child to execute the command, and then reads the next command when the child terminates. To wait for the child to finish, the parent executes a waitpid system call, which just waits until the child terminates (any child if more than one exists). Waitpid can wait for a specific child, or for any old child by setting the first parameter to −1. When waitpid completes, the address pointed to by the second parameter, *statloc*, will be set to the child's exit status (normal or abnormal termination and exit value). Various options are also provided, specified by the third parameter.

Now consider how fork is used by the shell. When a command is typed, the shell forks off a new process. This child process must execute the user command. It does this by using the execve system call, which causes its entire core image to be replaced by the file named in its first parameter. (Actually, the system call itself is exec, but several different library procedures call it with different parameters and slightly different names. We will treat these as system calls here.) A highly simplified shell illustrating the use of fork, waitpid, and execve is shown in Fig. 1-19.

In the most general case, execve has three parameters: the name of the file to be executed, a pointer to the argument array, and a pointer to the environment array. These will be described shortly. Various library routines, including *execl*, *execv*, *execle*, and *execve*, are provided to allow the parameters to be omitted or

Spotify USA Inc. Exhibit 1009 Page 68

```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
    type_prompt( );                             /* display prompt on the screen */
    read_command(command, parameters);          /* read input from terminal */

    if (fork( ) != 0) {                         /* fork off child process */
        /* Parent code. */
        waitpid(-1, &status, 0);                /* wait for child to exit */
    } else {
        /* Child code. */
        execve(command, parameters, 0);         /* execute command */
    }
}
```

Figure 1-19. A stripped-down shell. Throughout this book, *TRUE* is assumed to be defined as 1.

specified in various ways. Throughout this book we will use the name exec to represent the system call invoked by all of these.

Let us consider the case of a command such as

```
cp file1 file2
```

used to copy *file1* to *file2*. After the shell has forked, the child process locates and executes the file *cp* and passes to it the names of the source and target files.

The main program of *cp* (and main program of most other C programs) contains the declaration

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

where *argc* is a count of the number of items on the command line, including the program name. For the example above, *argc* is 3.

The second parameter, *argv*, is a pointer to an array. Element *i* of that array is a pointer to the *i*-th string on the command line. In our example, *argv*[0] would point to the string "cp", *argv*[1] would point to the string "file1" and *argv*[2] would point to the string "file2".

The third parameter of *main*, *envp*, is a pointer to the environment, an array of strings containing assignments of the form *name* = *value* used to pass information such as the terminal type and home directory name to a program. In Fig. 1-19, no environment is passed to the child, so the third parameter of *execve* is a zero.

If exec seems complicated, do not despair; it is (semantically) the most complex of all the POSIX system calls. All the other ones are much simpler. As an example of a simple one, consider exit, which processes should use when they are finished executing. It has one parameter, the exit status (0 to 255), which is returned to the parent via *statloc* in the waitpid system call.

Spotify USA Inc. Exhibit 1009 Page 69

Processes in UNIX have their memory divided up into three segments: the **text segment** (i.e., the program code), the **data segment** (i.e., the variables), and the **stack segment**. The data segment grows upward and the stack grows downward, as shown in Fig. 1-20. Between them is a gap of unused address space. The stack grows into the gap automatically, as needed, but expansion of the data segment is done explicitly by using a system call, brk, which specifies the new address where the data segment is to end. This call, however, is not defined by the POSIX standard, since programmers are encouraged to use the *malloc* library procedure for dynamically allocating storage, and the underlying implementation of *malloc* was not thought to be a suitable subject for standardization since few programmers use it directly.

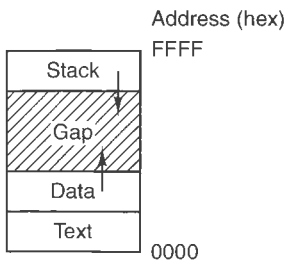


Figure 1-20. Processes have three segments: text, data, and stack.

1.6.2 System Calls for File Management

Many system calls relate to the file system. In this section we will look at calls that operate on individual files; in the next one we will examine those that involve directories or the file system as a whole.

To read or write a file, the file must first be opened using open. This call specifies the file name to be opened, either as an absolute path name or relative to the working directory, and a code of *O_RDONLY*, *O_WRONLY*, or *O_RDWR*, meaning open for reading, writing, or both. To create a new file, *O_CREAT* is used. The file descriptor returned can then be used for reading or writing. Afterward, the file can be closed by close, which makes the file descriptor available for reuse on a subsequent open.

The most heavily used calls are undoubtedly read and write. We saw read earlier. Write has the same parameters.

Although most programs read and write files sequentially, for some applications programs need to be able to access any part of a file at random. Associated with each file is a pointer that indicates the current position in the file. When reading (writing) sequentially, it normally points to the next byte to be read (written). The lseek call changes the value of the position pointer, so that subsequent calls to read or write can begin anywhere in the file.

Spotify USA Inc. Exhibit 1009 Page 70

Lseek has three parameters: the first is the file descriptor for the file, the second is a file position, and the third tells whether the file position is relative to the beginning of the file, the current position, or the end of the file. The value returned by lseek is the absolute position in the file after changing the pointer.

For each file, UNIX keeps track of the file mode (regular file, special file, directory, and so on), size, time of last modification, and other information. Programs can ask to see this information via the stat system call. The first parameter specifies the file to be inspected; the second one is a pointer to a structure where the information is to be put.

1.6.3 System Calls for Directory Management

In this section we will look at some system calls that relate more to directories or the file system as a whole, rather than just to one specific file as in the previous section. The first two calls, mkdir and rmdir, create and remove empty directories, respectively. The next call is link. Its purpose is to allow the same file to appear under two or more names, often in different directories. A typical use is to allow several members of the same programming team to share a common file, with each of them having the file appear in his own directory, possibly under different names. Sharing a file is not the same as giving every team member a private copy, because having a shared file means that changes that any member of the team makes are instantly visible to the other members—there is only one file. When copies are made of a file, subsequent changes made to one copy do not affect the other ones.

To see how link works, consider the situation of Fig. 1-21(a). Here are two users, *ast* and *jim*, each having their own directories with some files. If *ast* now executes a program containing the system call

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

the file *memo* in *jim*'s directory is now entered into *ast*'s directory under the name *note*. Thereafter, */usr/jim/memo* and */usr/ast/note* refer to the same file. As an aside, whether user directories are kept in */usr*, */user*, */home*, or somewhere else is simply a decision made by the local system administrator.

Understanding how link works will probably make it clearer what it does. Every file in UNIX has a unique number, its i-number, that identifies it. This i-number is an index into a table of **i-nodes**, one per file, telling who owns the file, where its disk blocks are, and so on. A directory is simply a file containing a set of (i-number, ASCII name) pairs. In the first versions of UNIX, each directory entry was 16 bytes—2 bytes for the i-number and 14 bytes for the name. Now a more complicated structure is needed to support long file names, but conceptually a directory is still a set of (i-number, ASCII name) pairs. In Fig. 1-21, *mail* has i-number 16, and so on. What link does is simply create a new directory entry with a (possibly new) name, using the i-number of an existing file. In Fig. 1-21(b), two

Spotify USA Inc. Exhibit 1009 Page 71

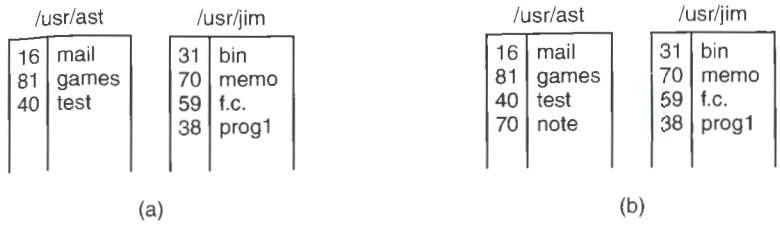


Figure 1-21. (a) Two directories before linking */usr/jim/memo* to ast's directory. (b) The same directories after linking.

entries have the same i-number (70) and thus refer to the same file. If either one is later removed, using the unlink system call, the other one remains. If both are removed, UNIX sees that no entries to the file exist (a field in the i-node keeps track of the number of directory entries pointing to the file), so the file is removed from the disk.

As we have mentioned earlier, the mount system call allows two file systems to be merged into one. A common situation is to have the root file system containing the binary (executable) versions of the common commands and other heavily used files, on a hard disk. The user can then insert a floppy disk with files to be read into the floppy disk drive.

By executing the mount system call, the floppy disk file system can be attached to the root file system, as shown in Fig. 1-22. A typical statement in C to perform the mount is

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

where the first parameter is the name of a block special file for drive 0, the second parameter is the place in the tree where it is to be mounted, and the third parameter tells whether the file system is to be mounted read-write or read-only.

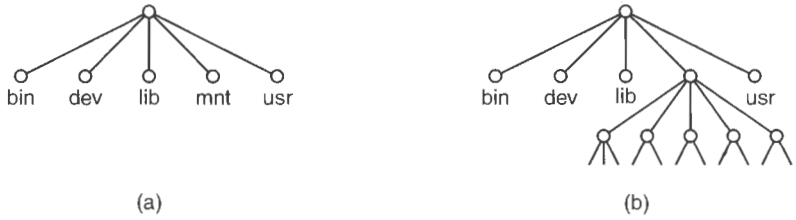


Figure 1-22. (a) File system before the mount. (b) File system after the mount.

After the mount call, a file on drive 0 can be accessed by just using its path from the root directory or the working directory, without regard to which drive it is on. In fact, second, third, and fourth drives can also be mounted anywhere in the tree. The mount call makes it possible to integrate removable media into a

Spotify USA Inc. Exhibit 1009 Page 72

single integrated file hierarchy, without having to worry about which device a file is on. Although this example involves floppy disks, hard disks or portions of hard disks (often called **partitions** or **minor devices**) can also be mounted this way. When a file system is no longer needed, it can be unmounted with the umount system call.

1.6.4 Miscellaneous System Calls

A variety of other system calls exist as well. We will look at just four of them here. The chdir call changes the current working directory. After the call

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

an open on the file *xyz* will open */usr/ast/test/xyz*. The concept of a working directory eliminates the need for typing (long) absolute path names all the time.

In UNIX every file has a mode used for protection. The mode includes the read-write-execute bits for the owner, group, and others. The chmod system call makes it possible to change the mode of a file. For example, to make a file read-only by everyone except the owner, one could execute

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

The kill system call is the way users and user processes send signals. If a process is prepared to catch a particular signal, then when it arrives, a signal handler is run. If the process is not prepared to handle a signal, then its arrival kills the process (hence the name of the call).

POSIX defines several procedures for dealing with time. For example, time just returns the current time in seconds, with 0 corresponding to Jan. 1, 1970 at midnight (just as the day was starting, not ending). On computers with 32-bit words, the maximum value time can return is $2^{32} - 1$ seconds (assuming an unsigned integer is used). This value corresponds to a little over 136 years. Thus in the year 2106, 32-bit UNIX systems will go berserk, imitating the famous Y2K problem. If you currently have a 32-bit UNIX system, you are advised to trade it in for a 64-bit one sometime before the year 2106.

1.6.5 The Windows Win32 API

So far we have focused primarily on UNIX. Now it is time to look briefly at Windows. Windows and UNIX differ in a fundamental way in their respective programming models. A UNIX program consists of code that does something or other, making system calls to have certain services performed. In contrast, a Windows program is normally event driven. The main program waits for some event to happen, then calls a procedure to handle it. Typical events are keys being struck, the mouse being moved, a mouse button being pushed, or a floppy disk inserted. Handlers are then called to process the event, update the screen and

update the internal program state. All in all, this leads to a somewhat different style of programming than with UNIX, but since the focus of this book is on operating system function and structure, these different programming models will not concern us much more.

Of course, Windows also has system calls. With UNIX, there is almost a 1-to-1 relationship between the system calls (e.g., read) and the library procedures (e.g., *read*) used to invoke the system calls. In other words, for each system call, there is roughly one library procedure that is called to invoke it, as indicated in Fig. 1-17. Furthermore, POSIX has only about 100 procedure calls.

With Windows, the situation is radically different. To start with, the library calls and the actual system calls are highly decoupled. Microsoft has defined a set of procedures, called the **Win32 API** (**Application Program Interface**) that programmers are expected to use to get operating system services. This interface is (partially) supported on all versions of Windows since Windows 95. By decoupling the interface from the actual system calls, Microsoft retains the ability to change the actual system calls in time (even from release to release) without invalidating existing programs. What actually constitutes Win32 is also slightly ambiguous since Windows 2000 has many new calls that were not previously available. In this section, Win32 means the interface supported by all versions of Windows.

The number of Win32 API calls is extremely large, numbering in the thousands. Furthermore, while many of them do invoke system calls, a substantial number are carried out entirely in user space. As a consequence, with Windows it is impossible to see what is a system call (i.e., performed by the kernel) and what is simply a user-space library call. In fact, what is a system call in one version of Windows may be done in user space in a different version, and vice versa. When we discuss the Windows system calls in this book, we will use the Win32 procedures (where appropriate) since Microsoft guarantees that these will be stable over time. But it is worth remembering that not all of them are true system calls (i.e., traps to the kernel).

Another complication is that in UNIX, the GUI (e.g., X Windows and Motif) runs entirely in user space, so the only system calls needed for writing on the screen are write and a few other minor ones. Of course, there are a large number of calls to X Windows and the GUI, but these are not system calls in any sense.

In contrast, the Win32 API has a huge number of calls for managing windows, geometric figures, text, fonts, scrollbars, dialog boxes, menus, and other features of the GUI. To the extent that the graphics subsystem runs in the kernel (true on some versions of Windows but not on all), these are system calls; otherwise they are just library calls. Should we discuss these calls in this book or not? Since they are not really related to the function of an operating system, we have decided not to, even though they may be carried out by the kernel. Readers interested in the Win32 API should consult one of the many books on the subject, for example (Hart, 1997; Rector and Newcomer, 1997; and Simon, 1997).

Spotify USA Inc. Exhibit 1009 Page 74

Even introducing all the Win32 API calls here is out of the question, so we will restrict ourselves to those calls that roughly correspond to the functionality of the UNIX calls listed in Fig. 1-18. These are listed in Fig. 1-23.

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Figure 1-23. The Win32 API calls that roughly correspond to the UNIX calls of Fig. 1-18.

Let us now briefly go through the list of Fig. 1-23. CreateProcess creates a new process. It does the combined work of fork and execve in UNIX. It has many parameters specifying the properties of the newly created process. Windows does not have a process hierarchy as UNIX does so there is no concept of a parent process and a child process. After a process is created, the creator and createe are equals. WaitForSingleObject is used to wait for an event. Many possible events can be waited for. If the parameter specifies a process, then the caller waits for the specified process to exit, which is done using ExitProcess.

The next six calls operate on files and are functionally similar to their UNIX counterparts although they differ in the parameters and details. Still, files can be opened, closed, read, and written pretty much as in UNIX. The SetFilePointer and GetFileAttributesEx calls set the file position and get some of the file attributes.

Spotify USA Inc. Exhibit 1009 Page 75

Windows has directories and they are created with CreateDirectory and RemoveDirectory, respectively. There is also a notion of a current directory, set by SetCurrentDirectory. The current time is acquired using GetLocalTime.

The Win32 interface does not have links to files, mounted file systems, security, or signals, so the calls corresponding to the UNIX ones do not exist. Of course, Win32 has a huge number of other calls that UNIX does not have, especially for managing the GUI. And Windows 2000 has an elaborate security system and also supports file links.

One last note about Win32 is perhaps worth making. Win32 is not a terribly uniform or consistent interface. The main culprit here was the need to be backward compatible with the previous 16-bit interface used in Windows 3.x.

1.7 OPERATING SYSTEM STRUCTURE

Now that we have seen what operating systems look like on the outside (i.e., the programmer's interface), it is time to take a look inside. In the following sections, we will examine five different structures that have been tried, in order to get some idea of the spectrum of possibilities. These are by no means exhaustive, but they give an idea of some designs that have been tried in practice. The five designs are monolithic systems, layered systems, virtual machines, exokernels, and client-server systems.

1.7.1 Monolithic Systems

By far the most common organization, this approach might well be subtitled "The Big Mess." The structure is that there is no structure. The operating system is written as a collection of procedures, each of which can call any of the other ones whenever it needs to. When this technique is used, each procedure in the system has a well-defined interface in terms of parameters and results, and each one is free to call any other one, if the latter provides some useful computation that the former needs.

To construct the actual object program of the operating system when this approach is used, one first compiles all the individual procedures, or files containing the procedures, and then binds them all together into a single object file using the system linker. In terms of information hiding, there is essentially none—every procedure is visible to every other procedure (as opposed to a structure containing modules or packages, in which much of the information is hidden away inside modules, and only the officially designated entry points can be called from outside the module).

Even in monolithic systems, however, it is possible to have at least a little structure. The services (system calls) provided by the operating system are requested by putting the parameters in a well-defined place (e.g., on the stack) and

Spotify USA Inc. Exhibit 1009 Page 76

then executing a trap instruction. This instruction switches the machine from user mode to kernel mode and transfers control to the operating system, shown as step 6 in Fig. 1-17. The operating system then fetches the parameters and determines which system call is to be carried out. After that, it indexes into a table that contains in slot *k* a pointer to the procedure that carries out system call *k* (step 7 in Fig. 1-17).

This organization suggests a basic structure for the operating system:

1. A main program that invokes the requested service procedure.
2. A set of service procedures that carry out the system calls.
3. A set of utility procedures that help the service procedures.

In this model, for each system call there is one service procedure that takes care of it. The utility procedures do things that are needed by several service procedures, such as fetching data from user programs. This division of the procedures into three layers is shown in Fig. 1-24.

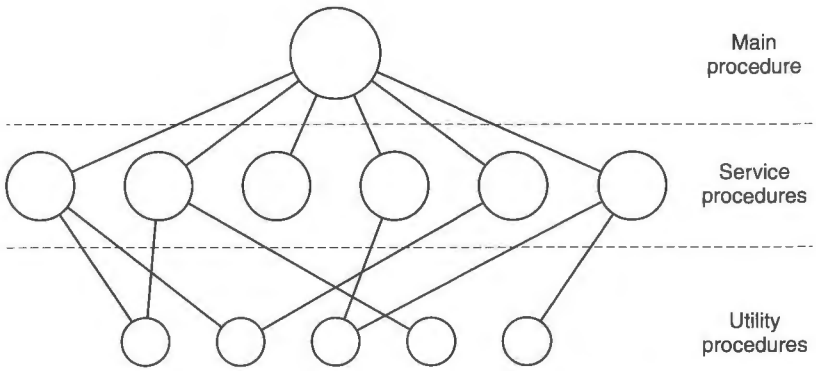


Figure 1-24. A simple structuring model for a monolithic system.

1.7.2 Layered Systems

A generalization of the approach of Fig. 1-24 is to organize the operating system as a hierarchy of layers, each one constructed upon the one below it. The first system constructed in this way was the THE system built at the Technische Hogeschool Eindhoven in the Netherlands by E. W. Dijkstra (1968) and his students. The THE system was a simple batch system for a Dutch computer, the Electrologica X8, which had 32K of 27-bit words (bits were expensive back then).

The system had 6 layers, as shown in Fig. 1-25. Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired. Above layer 0, the system consisted of sequential processes, each of

Spotify USA Inc. Exhibit 1009 Page 77

which could be programmed without having to worry about the fact that multiple processes were running on a single processor. In other words, layer 0 provided the basic multiprogramming of the CPU.

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Figure 1-25. Structure of the THE operating system.

Layer 1 did the memory management. It allocated space for processes in main memory and on a 512K word drum used for holding parts of processes (pages) for which there was no room in main memory. Above layer 1, processes did not have to worry about whether they were in memory or on the drum; the layer 1 software took care of making sure pages were brought into memory whenever they were needed.

Layer 2 handled communication between each process and the operator console. Above this layer each process effectively had its own operator console. Layer 3 took care of managing the I/O devices and buffering the information streams to and from them. Above layer 3 each process could deal with abstract I/O devices with nice properties, instead of real devices with many peculiarities. Layer 4 was where the user programs were found. They did not have to worry about process, memory, console, or I/O management. The system operator process was located in layer 5.

A further generalization of the layering concept was present in the MULTICS system. Instead of layers, MULTICS was described as having a series of concentric rings, with the inner ones being more privileged than the outer ones (which is effectively the same thing). When a procedure in an outer ring wanted to call a procedure in an inner ring, it had to make the equivalent of a system call, that is, a TRAP instruction whose parameters were carefully checked for validity before allowing the call to proceed. Although the entire operating system was part of the address space of each user process in MULTICS, the hardware made it possible to designate individual procedures (memory segments, actually) as protected against reading, writing, or executing.

Whereas the THE layering scheme was really only a design aid, because all the parts of the system were ultimately linked together into a single object program, in MULTICS, the ring mechanism was very much present at run time and enforced by the hardware. The advantage of the ring mechanism is that it can easily be extended to structure user subsystems. For example, a professor could

Spotify USA Inc. Exhibit 1009 Page 78

write a program to test and grade student programs and run this program in ring n, with the student programs running in ring $n + 1$ so that they could not change their grades.

1.7.3 Virtual Machines

The initial releases of OS/360 were strictly batch systems. Nevertheless, many 360 users wanted to have timesharing, so various groups, both inside and outside IBM decided to write timesharing systems for it. The official IBM timesharing system, TSS/360, was delivered late, and when it finally arrived it was so big and slow that few sites converted to it. It was eventually abandoned after its development had consumed some $50 million (Graham, 1970). But a group at IBM's Scientific Center in Cambridge, Massachusetts, produced a radically different system that IBM eventually accepted as a product, and which is now widely used on its remaining mainframes.

This system, originally called CP/CMS and later renamed VM/370 (Seawright and MacKinnon, 1979), was based on an astute observation: a timesharing system provides (1) multiprogramming and (2) an extended machine with a more convenient interface than the bare hardware. The essence of VM/370 is to completely separate these two functions.

The heart of the system, known as the **virtual machine monitor**, runs on the bare hardware and does the multiprogramming, providing not one, but several virtual machines to the next layer up, as shown in Fig. 1-26. However, unlike all other operating systems, these virtual machines are not extended machines, with files and other nice features. Instead, they are *exact* copies of the bare hardware, including kernel/user mode, I/O, interrupts, and everything else the real machine has.

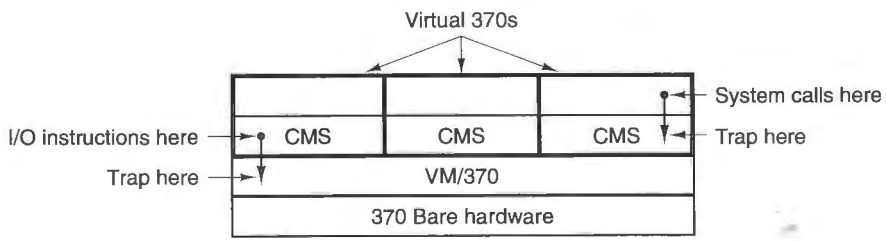


Figure 1-26. The structure of VM/370 with CMS.

Because each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the bare hardware. Different virtual machines can, and frequently do, run different operating systems. Some run one of the descendants of OS/360 for batch or transaction processing, while other ones run a single-user, interactive system called **CMS** (**Conversational Monitor System**) for interactive timesharing users.

Spotify USA Inc. Exhibit 1009 Page 79

When a CMS program executes a system call, the call is trapped to the operating system in its own virtual machine, not to VM/370, just as it would if it were running on a real machine instead of a virtual one. CMS then issues the normal hardware I/O instructions for reading its virtual disk or whatever is needed to carry out the call. These I/O instructions are trapped by VM/370, which then performs them as part of its simulation of the real hardware. By completely separating the functions of multiprogramming and providing an extended machine, each of the pieces can be much simpler, more flexible, and easier to maintain.

The idea of a virtual machine is heavily used nowadays in a different context: running old MS-DOS programs on a Pentium (or other 32-bit Intel CPU). When designing the Pentium and its software, both Intel and Microsoft realized that there would be a big demand for running old software on new hardware. For this reason, Intel provided a virtual 8086 mode on the Pentium. In this mode, the machine acts like an 8086 (which is identical to an 8088 from a software point of view), including 16-bit addressing with a 1-MB limit.

This mode is used by Windows and other operating systems for running MS-DOS programs. These programs are started up in virtual 8086 mode. As long as they execute normal instructions, they run on the bare hardware. However, when a program tries to trap to the operating system to make a system call, or tries to do protected I/O directly, a trap to the virtual machine monitor occurs.

Two variants on this design are possible. In the first one, MS-DOS itself is loaded into the virtual 8086's address space, so the virtual machine monitor just reflects the trap back to MS-DOS, just as would happen on a real 8086. When MS-DOS later tries to do the I/O itself, that operation is caught and carried out by the virtual machine monitor.

In the other variant, the virtual machine monitor just catches the first trap and does the I/O itself, since it knows what all the MS-DOS system calls are and thus knows what each trap is supposed to do. This variant is less pure than the first one, since it only emulates MS-DOS correctly, and not other operating systems, as the first one does. On the other hand, it is much faster, since it saves the trouble of starting up MS-DOS to do the I/O. A further disadvantage of actually running MS-DOS in virtual 8086 mode is that MS-DOS fiddles around with the interrupt enable/disable bit quite a lot, all of which must be emulated at considerable cost.

It is worth noting that neither of these approaches are really the same as VM/370, since the machine being emulated is not a full Pentium, but only an 8086. With the VM/370 system, it is possible to run VM/370, itself, in the virtual machine. With the Pentium, it is not possible to run, say, Windows in the virtual 8086 because no version of Windows runs on an 8086; a 286 is the minimum for even the oldest version, and 286 emulation is not provided (let alone Pentium emulation). However, by modifying the Windows binary slightly, this emulation is possible and even available in commercial products.

Another area where virtual machines are used, but in a somewhat different way, is for running Java programs. When Sun Microsystems invented the Java

Spotify USA Inc. Exhibit 1009 Page 80

programming language, it also invented a virtual machine (i.e., a computer architecture) called the **JVM** (**Java Virtual Machine**). The Java compiler produces code for JVM, which then typically is executed by a software JVM interpreter. The advantage of this approach is that the JVM code can be shipped over the Internet to any computer that has a JVM interpreter and run there. If the compiler had produced SPARC or Pentium binary programs, for example, they could not have been shipped and run anywhere as easily. (Of course, Sun could have produced a compiler that produced SPARC binaries and then distributed a SPARC interpreter, but JVM is a much simpler architecture to interpret.) Another advantage of using JVM is that if the interpreter is implemented properly, which is not completely trivial, incoming JVM programs can be checked for safety and then executed in a protected environment so they cannot steal data or do any damage.

1.7.4 Exokernels

With VM/370, each user process gets an exact copy of the actual computer. With virtual 8086 mode on the Pentium, each user process gets an exact copy of a different computer. Going one step further, researchers at M.I.T. have built a system that gives each user a clone of the actual computer, but with a subset of the resources (Engler et al., 1995). Thus one virtual machine might get disk blocks 0 to 1023, the next one might get blocks 1024 to 2047, and so on.

At the bottom layer, running in kernel mode, is a program called the **exokernel**. Its job is to allocate resources to virtual machines and then check attempts to use them to make sure no machine is trying to use somebody else's resources. Each user-level virtual machine can run its own operating system, as on VM/370 and the Pentium virtual 8086s, except that each one is restricted to using only the resources it has asked for and been allocated.

The advantage of the exokernel scheme is that it saves a layer of mapping. In the other designs, each virtual machine thinks it has its own disk, with blocks running from 0 to some maximum, so the virtual machine monitor must maintain tables to remap disk addresses (and all other resources). With the exokernel, this remapping is not needed. The exokernel need only keep track of which virtual machine has been assigned which resource. This method still has the advantage of separating the multiprogramming (in the exokernel) from the user operating system code (in user space), but with less overhead, since all the exokernel has to do is keep the virtual machines out of each other's hair.

1.7.5 Client-Server Model

VM/370 gains much in simplicity by moving a large part of the traditional operating system code (implementing the extended machine) into a higher layer, CMS. Nevertheless, VM/370 itself is still a complex program because simulating a

Spotify USA Inc. Exhibit 1009 Page 81

number of virtual 370s in their entirety is not *that* simple (especially if you want to do it reasonably efficiently).

A trend in modern operating systems is to take the idea of moving code up into higher layers even further and remove as much as possible from kernel mode, leaving a minimal **microkernel**. The usual approach is to implement most of the operating system in user processes. To request a service, such as reading a block of a file, a user process (now known as the **client process**) sends the request to a **server process**, which then does the work and sends back the answer.

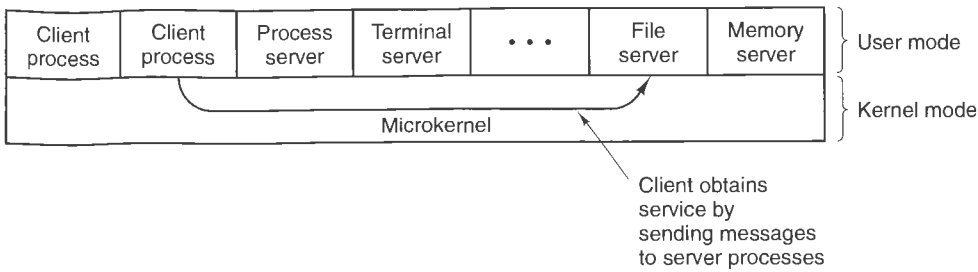


Figure 1-27. The client-server model.

In this model, shown in Fig. 1-27, all the kernel does is handle the communication between clients and servers. By splitting the operating system up into parts, each of which only handles one facet of the system, such as file service, process service, terminal service, or memory service, each part becomes small and manageable. Furthermore, because all the servers run as user-mode processes, and not in kernel mode, they do not have direct access to the hardware. As a consequence, if a bug in the file server is triggered, the file service may crash, but this will not usually bring the whole machine down.

Another advantage of the client-server model is its adaptability to use in distributed systems (see Fig. 1-28). If a client communicates with a server by sending it messages, the client need not know whether the message is handled locally in its own machine, or whether it was sent across a network to a server on a remote machine. As far as the client is concerned, the same thing happens in both cases: a request was sent and a reply came back.

The picture painted above of a kernel that handles only the transport of messages from clients to servers and back is not completely realistic. Some operating system functions (such as loading commands into the physical I/O device registers) are difficult, if not impossible, to do from user-space programs. There are two ways of dealing with this problem. One way is to have some critical server processes (e.g., I/O device drivers) actually run in kernel mode, with complete access to all the hardware, but still communicate with other processes using the normal message mechanism.

The other way is to build a minimal amount of **mechanism** into the kernel but leave the **policy** decisions up to servers in user space (Levin et al., 1975). For

Spotify USA Inc. Exhibit 1009 Page 82

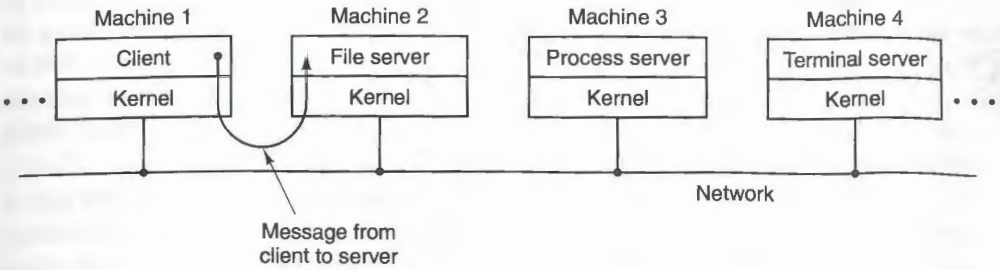


Figure 1-28. The client-server model in a distributed system.

example, the kernel might recognize that a message sent to a certain special address means to take the contents of that message and load it into the I/O device registers for some disk, to start a disk read. In this example, the kernel would not even inspect the bytes in the message to see if they were valid or meaningful; it would just blindly copy them into the disk's device registers. (Obviously, some scheme for limiting such messages to authorized processes only must be used.) The split between mechanism and policy is an important concept; it occurs again and again in operating systems in various contexts.

1.8 RESEARCH ON OPERATING SYSTEMS

Computer science is a rapidly advancing field and it is hard to predict where it is going. Researchers at universities and industrial research labs are constantly thinking up new ideas, some of which go nowhere but some of which become the cornerstone of future products and have massive impact on the industry and users. Telling which is which turns out to be easier to do in hindsight than in real time. Separating the wheat from the chaff is especially difficult because it often takes 20-30 years from idea to impact.

For example, when President Eisenhower set up the Dept. of Defense's Advanced Research Projects Agency (ARPA) in 1958, he was trying to keep the Army from killing the Navy and the Air Force over the Pentagon's research budget. He was not trying to invent the Internet. But one of the things ARPA did was fund some university research on the then-obscure concept of packet switching, which quickly led to the first experimental packet-switched network, the ARPANET. It went live in 1969. Before long, other ARPA-funded research networks were connected to the ARPANET, and the Internet was born. The Internet was then happily used by academic researchers for sending email to each other for 20 years. In the early 1990s, Tim Berners-Lee invented the World Wide Web at the CERN research lab in Geneva and Marc Andreesen wrote a graphical browser for it at the University of Illinois. All of a sudden the Internet was full of chatting teenagers. President Eisenhower is probably rolling over in his grave.

Spotify USA Inc. Exhibit 1009 Page 83

Research in operating systems has also led to dramatic changes in practical systems. As we discussed earlier, the first commercial computer systems were all batch systems, until M.I.T. invented interactive timesharing in the early 1960s. Computers were all text-based until Doug Engelbart invented the mouse and the graphical user interface at Stanford Research Institute in the late 1960s. Who knows what will come next?

In this section and in comparable sections throughout the book, we will take a brief look at some of the research in operating systems that has taken place during the past 5 to 10 years, just to give a flavor of what might be on the horizon. This introduction is certainly not comprehensive and is based largely on papers that have been published in the top research journals and conferences because these ideas have at least survived a rigorous peer review process in order to get published. Most of the papers cited in the research sections were published by either ACM, the IEEE Computer Society, or USENIX and are available over the Internet to (student) members of these organizations. For more information about these organizations and their digital libraries, see

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

Virtually all operating systems researchers realize that current operating systems are massive, inflexible, unreliable, insecure, and loaded with bugs, certain ones more than others (*names withheld here to protect the guilty*). Consequently, there is a lot of research on how to build flexible and dependable systems. Much of the research concerns microkernel systems. These systems have a minimal kernel, so there is a reasonable chance they can be made reliable and be debugged. They are also flexible because much of the real operating system runs as user-mode processes, and can thus be replaced or adapted easily, possibly even during execution. Typically, all the microkernel does is handle low-level resource management and message passing between the user processes.

The first generation microkernels, such as Amoeba (Tanenbaum et al., 1990), Chorus (Rozier et al., 1988), Mach (Accetta et al., 1986), and V (Cheriton, 1988), proved that these systems could be built and made to work. The second generation is trying to prove that they can not only work, but with high performance as well (Ford et al., 1996; Hartig et al., 1997; Liedtke 1995, 1996; Rawson 1997; and Zuberi et al., 1999). Based on published measurements, it appears that this goal has been achieved.

Much kernel research is focused nowadays on building extensible operating systems. These are typically microkernel systems with the ability to extend or customize them in some direction. Some examples are Fluke (Ford et al., 1997), Paramecium (Van Doorn et al., 1995), SPIN (Bershad et al., 1995b), and Vino (Seltzer et al., 1996). Some researchers are also looking at how to extend existing

Spotify USA Inc. Exhibit 1009 Page 84

systems (Ghormley et al., 1998). Many of these systems allow users to add their own code to the kernel, which brings up the obvious problem of how to allow user extensions in a secure way. Techniques include interpreting the extensions, restricting them to code sandboxes, using type-safe languages, and code signing (Grimm and Bershad, 1997; and Small and Seltzer, 1998). Druschel et al. (1997) present a dissenting view, saying that too much effort is going into security for user-extendable systems. In their view, researchers should figure out which extensions are useful and then just make those a normal part of the kernel, without the ability to have users extend the kernel on the fly.

Although one approach to eliminating bloated, buggy, unreliable operating systems is to make them smaller, a more radical one is to eliminate the operating system altogether. This approach is being taken by the group of Kaashoek at M.I.T. in their Exokernel research. Here the idea is to have a thin layer of software running on the bare metal, whose only job is to securely allocate the hardware resources among the users. For example, it must decide who gets to use which part of the disk and where incoming network packets should be delivered. Everything else is up to user-level processes, making it possible to build both general-purpose and highly-specialized operating systems (Engler and Kaashoek, 1995; Engler et al., 1995; and Kaashoek et al., 1997).

1.9 OUTLINE OF THE REST OF THIS BOOK

We have now completed our introduction and bird's-eye view of the operating system. It is time to get down to the details. Chapter 2 is about processes. It discusses their properties and how they communicate with one another. It also gives a number of detailed examples of how interprocess communication works and how to avoid some of the pitfalls.

Chapter 3 is about deadlocks. We briefly showed what deadlocks are in this chapter, but there is much more to say. Ways to prevent or avoid them are discussed.

In Chap. 4 we will study memory management in detail. The important topic of virtual memory will be examined, along with closely related concepts such as paging and segmentation.

Input/Output is covered in Chap. 5. The concepts of device independence and device dependence will be looked at. Several important devices, including disks, keyboards, and displays, will be used as examples.

Then, in Chap. 6, we come to the all-important topic of file systems. To a considerable extent, what the user sees is largely the file system. We will look at both the file system interface and the file system implementation.

At this point we will have completed our study of the basic principles of single-CPU operating systems. However, there is more to say, especially about advanced topics. In Chap. 7, we examine multimedia systems, which have a num-

Spotify USA Inc. Exhibit 1009 Page 85

ber of properties and requirements that differ from conventional operating systems. Among other items, scheduling and the file system are affected by the nature of multimedia. Another advanced topic is multiple processor systems, including multiprocessors, parallel computers, and distributed systems. These subjects are covered in Chap. 8.

A hugely important subject is operating system security, which is covered in Chap 9. Among the topics discussed in this chapter are threats (e.g., viruses and worms), protection mechanisms, and security models.

Next we have some case studies of real operating systems. These are UNIX (Chap. 10) and Windows 2000 (Chap. 11). The book concludes with some thoughts about operating system design in Chap. 12.

1.10 METRIC UNITS

To avoid any confusion, it is worth stating explicitly that in this book, as in computer science in general, metric units are used instead of traditional English units (the furlong-stone-fortnight system). The principal metric prefixes are listed in Fig. 1-29. The prefixes are typically abbreviated by their first letters, with the units greater than 1 capitalized. Thus a 1-TB database occupies 10^{12} bytes of storage and a 100 psec (or 100 ps) clock ticks every 10^{-10} seconds. Since milli and micro both begin with the letter "m," a choice had to be made. Normally, "m" is for milli and "μ" (the Greek letter mu) is for micro.

Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	Yotta

Figure 1-29. The principal metric prefixes.

It is also worth pointing out that for measuring memory sizes, in common industry practice, the units have slightly different meanings. There Kilo means 2^{10} (1024) rather than 10^{3} (1000) because memories are always a power of two. Thus a 1-KB memory contains 1024 bytes, not 1000 bytes. Similarly, a 1-MB memory contains 2^{20} (1,048,576) bytes and a 1-GB memory contains 2^{30} (1,073,741,824) bytes. However, a 1-Kbps communication line transmits 1000 bits per second and a 10-Mbps LAN runs at 10,000,000 bits/sec because these speeds are not powers of two. Unfortunately, many people tend to mix up these

Spotify USA Inc. Exhibit 1009 Page 86

two systems, especially for disk sizes. To avoid ambiguity, in this book, we will use the symbols KB, MB, and GB for 2^{10}, 2^{20}, and 2^{30} bytes respectively, and the symbols Kbps, Mbps, and Gbps for 10^3, 10^6 and 10^9 bits/sec, respectively.

1.11 SUMMARY

Operating systems can be viewed from two viewpoints: resource managers and extended machines. In the resource manager view, the operating system's job is to manage the different parts of the system efficiently. In the extended machine view, the job of the system is to provide the users with a virtual machine that is more convenient to use than the actual machine.

Operating systems have a long history, starting from the days when they replaced the operator, to modern multiprogramming systems. Highlights include early batch systems, multiprogramming systems, and personal computer systems.

Since operating systems interact closely with the hardware, some knowledge of computer hardware is useful to understanding them. Computers are built up of processors, memories, and I/O devices. These parts are connected by buses.

The basic concepts on which all operating systems are built are processes, memory management, I/O management, the file system, and security. Each of these will be treated in a subsequent chapter.

The heart of any operating system is the set of system calls that it can handle. These tell what the operating system really does. For UNIX, we have looked at four groups of system calls. The first group of system calls relates to process creation and termination. The second group is for reading and writing files. The third group is for directory management. The fourth group contains miscellaneous calls.

Operating systems can be structured in several ways. The most common ones are as a monolithic system, a hierarchy of layers, a virtual machine system, an exokernel, or using the client-server model.

PROBLEMS

1. What are the two main functions of an operating system?
2. What is multiprogramming?
3. What is spooling? Do you think that advanced personal computers will have spooling as a standard feature in the future?
4. On early computers, every byte of data read or written was directly handled by the CPU (i.e., there was no DMA). What implications does this organization have for multiprogramming?

Spotify USA Inc. Exhibit 1009 Page 87

5. Why was timesharing not widespread on second-generation computers?

6. The family of computers idea was introduced in the 1960s with the IBM System/360 mainframes. Is this idea now dead as a doornail or does it live on?

7. One reason GUIs were initially slow to be adopted was the cost of the hardware needed to support them. How much video RAM is needed to support a 25 line × 80 row character monochrome text screen? How much for a 1024 × 768 pixel 24-bit color bitmap? What was the cost of this RAM at 1980 prices ($5/KB)? How much is it now?

8. Which of the following instructions should be allowed only in kernel mode?

   (a) Disable all interrupts.
   (b) Read the time-of-day clock.
   (c) Set the time-of-day clock.
   (d) Change the memory map.

9. List some differences between personal computer operating systems and mainframe operating systems.

10. A computer has a pipeline with four stages. Each stage takes the same time to do its work, namely, 1 nsec. How many instructions per second can this machine execute?

11. An alert reviewer notices a consistent spelling error in the manuscript of an operating systems textbook that is about to go to press. The book has approximately 700 pages, each with 50 lines of 80 characters each. How long will it take to electronically scan the text for the case of the master copy being in each of the levels of memory of Fig. 1-7? For internal storage methods, consider that the access time given is per character, for disk devices assume the time is per block of 1024 characters, and for tape assume the time given is to the start of the data with subsequent access at the same speed as disk access.

12. In Fig. 1-9, the MMU compares the incoming (virtual) address to the limit register, causing a fault if it is too large. An alternative design would be to first add the virtual address to the base register and then compare the result to the (physical) address in the limit register. Are the two methods logically equivalent? Are they equivalent in performance?

13. When a user program makes a system call to read or write a disk file, it provides an indication of which file it wants, a pointer to the data buffer, and the count. Control is then transferred to the operating system, which calls the appropriate driver. Suppose that the driver starts the disk and terminates until an interrupt occurs. In the case of reading from the disk, obviously the caller will have to be blocked (because there are no data for it). What about the case of writing to the disk? Need the caller be blocking awaiting completion of the disk transfer?

14. What is the key difference between a trap and an interrupt?

15. A computer uses the relocation scheme of Fig. 1-9(a). A program is 10,000 bytes long and is loaded at address 40,000. What values do the *base* and *limit* register get according to the scheme described in the text?

Spotify USA Inc. Exhibit 1009 Page 88

16. Why is the process table needed in a timesharing system? Is it also needed in personal computer systems in which only one process exists, that process taking over the entire machine until it is finished?

17. Is there any reason why you might want to mount a file system on a nonempty directory? If so, what is it?

18. For each of the following system calls, give a condition that causes it to fail: fork, exec, and unlink.

19. Can the

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

call return any value in *count* other than *nbytes*? If so, why?

20. A file whose file descriptor is *fd* contains the following sequence of bytes: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. The following system calls are made:

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

where the lseek call makes a seek to byte 3 of the file. What does *buffer* contain after the read has completed?

21. What is the essential difference between a block special file and a character special file?

22. In the example given in Fig. 1-17, the library procedure is called *read* and the system call itself is called read. Is it essential that both of these have the same name? If not, which one is more important?

23. The client-server model is popular in distributed systems. Can it also be used in a single-computer system?

24. To a programmer, a system call looks like any other call to a library procedure. Is it important that a programmer know which library procedures result in system calls? Under what circumstances and why?

25. Figure 1-23 shows that a number of UNIX system calls have no Win32 API equivalents. For each of the calls listed as having no Win32 equivalent, what are the consequences for a programmer of converting a UNIX program to run under Windows?

26. Here are some questions for practicing unit conversions:

(a) How long is a microyear in seconds?
(b) Micrometers are often called microns. How long is a gigamicron?
(c) How many bytes are there in a 1-TB memory?
(d) The mass of the earth is 6000 yottagrams. What is that in kilograms?

27. Write a shell that is similar to Fig. 1-19 but contains enough code that it actually works so you can test it. You might also add some features such as redirection of input and output, pipes, and background jobs.

28. If you have a personal UNIX-like system (Linux, MINIX, Free BSD, etc.) available that you can safely crash and reboot, write a shell script that attempts to create an

Spotify USA Inc. Exhibit 1009 Page 89

unlimited number of child processes and observe what happens. Before running the experiment, type sync to the shell to flush the file system buffers to disk to avoid ruining the file system. **Note**: Do not try this on a shared system without first getting permission from the system administrator. The consequences will be instantly obvious so you are likely to be caught and sanctions may follow.

29. Examine and try to interpret the contents of a UNIX-like or Windows directory with a tool like the UNIX *od* program or the MS-DOS *DEBUG* program. *Hint*: How you do this will depend upon what the OS allows. One trick that may work is to create a directory on a floppy disk with one operating system and then read the raw disk data using a different operating system that allows such access.

Spotify USA Inc. Exhibit 1009 Page 90

6

FILE SYSTEMS

All computer applications need to store and retrieve information. While a process is running, it can store a limited amount of information within its own address space. However, the storage capacity is restricted to the size of the virtual address space. For some applications this size is adequate, but for others, such as airline reservations, banking, or corporate record keeping, it is far too small.

A second problem with keeping information within a process' address space is that when the process terminates, the information is lost. For many applications, (e.g., for databases), the information must be retained for weeks, months, or even forever. Having it vanish when the process using it terminates is unacceptable. Furthermore, it must not go away when a computer crash kills the process.

A third problem is that it is frequently necessary for multiple processes to access (parts of) the information at the same time. If we have an online telephone directory stored inside the address space of a single process, only that process can access it. The way to solve this problem is to make the information itself independent of any one process.

Thus we have three essential requirements for long-term information storage:

1. It must be possible to store a very large amount of information.
2. The information must survive the termination of the process using it.
3. Multiple processes must be able to access the information concurrently.

The usual solution to all these problems is to store information on disks and other external media in units called **files**. Processes can then read them and write new

Spotify USA Inc. Exhibit 1009 Page 91

ones if need be. Information stored in files must be **persistent**, that is, not be affected by process creation and termination. A file should only disappear when its owner explicitly removes it.

Files are managed by the operating system. How they are structured, named, accessed, used, protected, and implemented are major topics in operating system design. As a whole, that part of the operating system dealing with files is known as the **file system** and is the subject of this chapter.

From the users' standpoint, the most important aspect of a file system is how it appears to them, that is, what constitutes a file, how files are named and protected, what operations are allowed on files, and so on. The details of whether linked lists or bitmaps are used to keep track of free storage and how many sectors there are in a logical block are of less interest, although they are of great importance to the designers of the file system. For this reason, we have structured the chapter as several sections. The first two are concerned with the user interface to files and directories, respectively. Then comes a detailed discussion of how the file system is implemented. Finally, we give some examples of real file systems.

6.1 FILES

In the following pages we will look at files from the user's point of view, that is, how they are used and what properties they have.

6.1.1 File Naming

Files are an abstraction mechanism. They provide a way to store information on the disk and read it back later. This must be done in such a way as to shield the user from the details of how and where the information is stored, and how the disks actually work.

Probably the most important characteristic of any abstraction mechanism is the way the objects being managed are named, so we will start our examination of file systems with the subject of file naming. When a process creates a file, it gives the file a name. When the process terminates, the file continues to exist and can be accessed by other processes using its name.

The exact rules for file naming vary somewhat from system to system, but all current operating systems allow strings of one to eight letters as legal file names. Thus *andrea*, *bruce*, and *cathy* are possible file names. Frequently digits and special characters are also permitted, so names like *2*, *urgent!*, and *Fig.2-14* are often valid as well. Many file systems support names as long as 255 characters.

Some file systems distinguish between upper and lower case letters, whereas others do not. UNIX falls in the first category; MS-DOS falls in the second. Thus a UNIX system can have all of the following as three distinct files: *maria*, *Maria*, and *MARIA*. In MS-DOS, all these names refer to the same file.

Spotify USA Inc. Exhibit 1009 Page 92

An aside on file systems is probably in order here. Windows 95 and Windows 98 both use the MS-DOS file system, and thus inherit many of its properties, such as how file names are constructed. In addition, Windows NT and Windows 2000 support the MS-DOS file system and thus also inherit its properties. However, the latter two systems also have a native file system (NTFS) that has different properties (such as file names in Unicode). In this chapter, when we refer to the Windows file system, we mean the MS-DOS file system, which is the only file system supported by all versions of Windows. We will discuss the Windows 2000 native file system in Chap. 11.

Many operating systems support two-part file names, with the two parts separated by a period, as in *prog.c*. The part following the period is called the **file extension** and usually indicates something about the file. In MS-DOS, for example, file names are 1 to 8 characters, plus an optional extension of 1 to 3 characters. In UNIX, the size of the extension, if any, is up to the user, and a file may even have two or more extensions, as in *prog.c.Z*, where *.Z* is commonly used to indicate that the file (*prog.c*) has been compressed using the Ziv-Lempel compression algorithm. Some of the more common file extensions and their meanings are shown in Fig. 6-1.

Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

Figure 6-1. Some typical file extensions.

In some systems (e.g., UNIX), file extensions are just conventions and are not enforced by the operating system. A file named *file.txt* might be some kind of text file, but that name is more to remind the owner than to convey any actual information to the computer. On the other hand, a C compiler may actually insist that files it is to compile end in *.c*, and it may refuse to compile them if they do not.

Spotify USA Inc. Exhibit 1009 Page 93

Conventions like this are especially useful when the same program can handle several different kinds of files. The C compiler, for example, can be given a list of several files to compile and link together, some of them C files and some of them assembly language files. The extension then becomes essential for the compiler to tell which are C files, which are assembly files, and which are other files.

In contrast, Windows is aware of the extensions and assigns meaning to them. Users (or processes) can register extensions with the operating system and specify for each one which program "owns" that extension. When a user double clicks on a file name, the program assigned to its file extension is launched with the file as parameter. For example, double clicking on *file.doc* starts Microsoft *Word* with *file.doc* as the initial file to edit.

6.1.2 File Structure

Files can be structured in any of several ways. Three common possibilities are depicted in Fig. 6-2. The file in Fig. 6-2(a) is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs. Both UNIX and Windows use this approach.

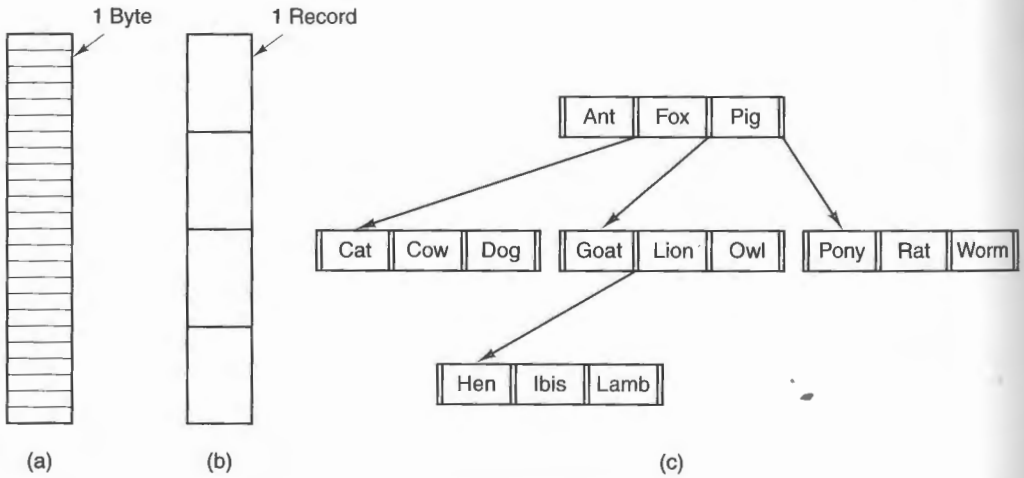


Figure 6-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

Having the operating system regard files as nothing more than byte sequences provides the maximum flexibility. User programs can put anything they want in their files and name them any way that is convenient. The operating system does

Spotify USA Inc. Exhibit 1009 Page 94

not help, but it also does not get in the way. For users who want to do unusual things, the latter can be very important.

The first step up in structure is shown in Fig. 6-2(b). In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record. As a historical note, in decades gone by, when the 80-column punched card was king, many (mainframe) operating systems based their file systems on files consisting of 80-character records, in effect, card images. These systems also supported files of 132-character records, which were intended for the line printer (which in those days were big chain printers having 132 columns). Programs read input in units of 80 characters and wrote it in units of 132 characters, although the final 52 could be spaces, of course. No current general-purpose system works this way.

The third kind of file structure is shown in Fig. 6-2(c). In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a **key** field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.

The basic operation here is not to get the "next" record, although that is also possible, but to get the record with a specific key. For the zoo file of Fig. 6-2(c), one could ask the system to get the record whose key is *pony*, for example, without worrying about its exact position in the file. Furthermore, new records can be added to the file, with the operating system, and not the user, deciding where to place them. This type of file is clearly quite different from the unstructured byte streams used in UNIX and Windows but is widely used on the large mainframe computers still used in some commercial data processing.

6.1.3 File Types

Many operating systems support several types of files. UNIX and Windows, for example, have regular files and directories. UNIX also has character and block special files. **Regular files** are the ones that contain user information. All the files of Fig. 6-2 are regular files. **Directories** are system files for maintaining the structure of the file system. We will study directories below. **Character special files** are related to input/output and used to model serial I/O devices such as terminals, printers, and networks. **Block special files** are used to model disks. In this chapter we will be primarily interested in regular files.

Regular files are generally either ASCII files or binary files. ASCII files consist of lines of text. In some systems each line is terminated by a carriage return character. In others, the line feed character is used. Some systems (e.g., MS-DOS) use both. Lines need not all be of the same length.

The great advantage of ASCII files is that they can be displayed and printed as is, and they can be edited with any text editor. Furthermore, if large numbers

Spotify USA Inc. Exhibit 1009 Page 95

of programs use ASCII files for input and output, it is easy to connect the output of one program to the input of another, as in shell pipelines. (The interprocess plumbing is not any easier, but interpreting the information certainly is if a standard convention, such as ASCII, is used for expressing it.)

Other files are binary files, which just means that they are not ASCII files. Listing them on the printer gives an incomprehensible listing full of what is apparently random junk. Usually, they have some internal structure known to programs that use them.

For example, in Fig. 6-3(a) we see a simple executable binary file taken from a version of UNIX. Although technically the file is just a sequence of bytes, the operating system will only execute a file if it has the proper format. It has five sections: header, text, data, relocation bits, and symbol table. The header starts with a so-called **magic number**, identifying the file as an executable file (to prevent the accidental execution of a file not in this format). Then come the sizes of the various pieces of the file, the address at which execution starts, and some flag bits. Following the header are the text and data of the program itself. These are loaded into memory and relocated using the relocation bits. The symbol table is used for debugging.

Our second example of a binary file is an archive, also from UNIX. It consists of a collection of library procedures (modules) compiled but not linked. Each one is prefaced by a header telling its name, creation date, owner, protection code, and size. Just as with the executable file, the module headers are full of binary numbers. Copying them to the printer would produce complete gibberish.

Every operating system must recognize at least one file type: its own executable file, but some recognize more. The old TOPS-20 system (for the DECsystem 20) went so far as to examine the creation time of any file to be executed. Then it located the source file and saw if the source had been modified since the binary was made. If it had been, it automatically recompiled the source. In UNIX terms, the *make* program had been built into the shell. The file extensions were mandatory so the operating system could tell which binary program was derived from which source.

Having strongly typed files like this causes problems whenever the user does anything that the system designers did not expect. Consider, as an example, a system in which program output files have extension *.dat* (data files). If a user writes a program formatter that reads a *.c* file (C program), transforms it (e.g., by converting it to a standard indentation layout), and then writes the transformed file as output, the output file will be of type *.dat*. If the user tries to offer this to the C compiler to compile it, the system will refuse because it has the wrong extension. Attempts to copy *file.dat* to *file.c* will be rejected by the system as invalid (to protect the user against mistakes).

While this kind of "user friendliness" may help novices, it drives experienced users up the wall since they have to devote considerable effort to circumventing the operating system's idea of what is reasonable and what is not.

Spotify USA Inc. Exhibit 1009 Page 96

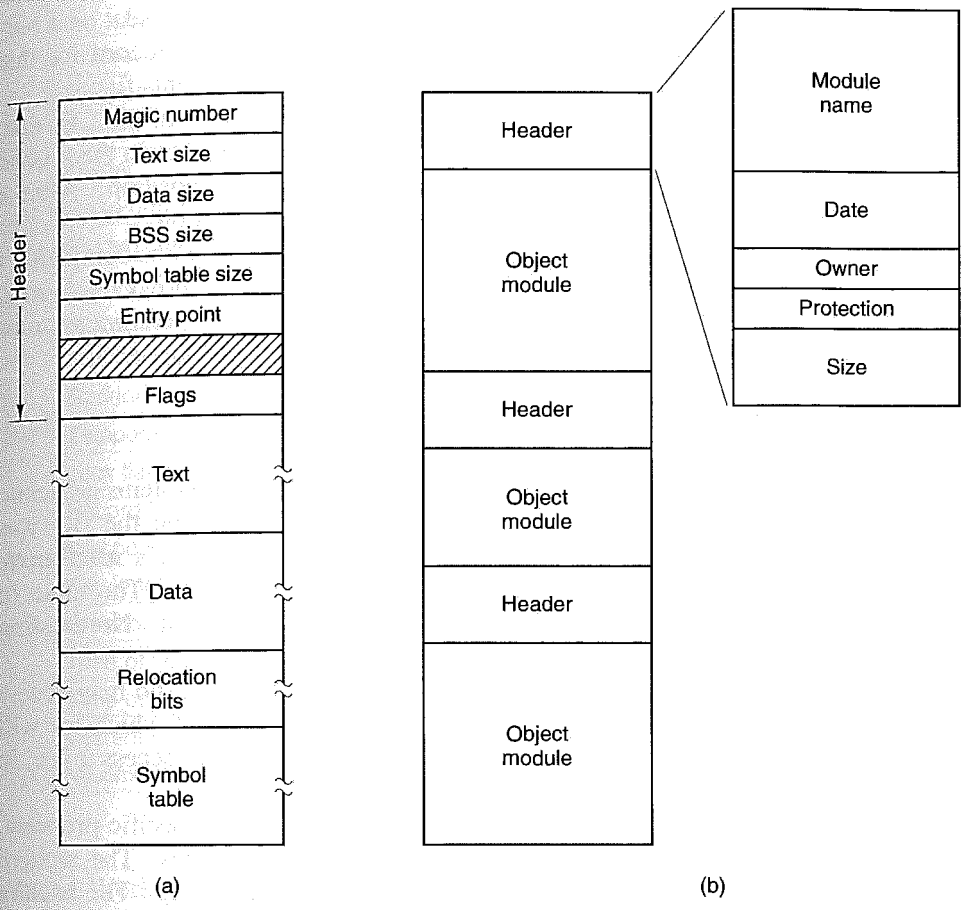


Figure 6-3. (a) An executable file. (b) An archive.

6.1.4 File Access

Early operating systems provided only one kind of file access: **sequential access**. In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order. Sequential files could be rewound, however, so they could be read as often as needed. Sequential files were convenient when the storage medium was magnetic tape, rather than disk.

When disks came into use for storing files, it became possible to read the bytes or records of a file out of order, or to access records by key, rather than by position. Files whose bytes or records can be read in any order are called **random access files**. They are required by many applications.

Spotify USA Inc. Exhibit 1009 Page 97

Random access files are essential for many applications, for example, database systems. If an airline customer calls up and wants to reserve a seat on a particular flight, the reservation program must be able to access the record for that flight without having to read the records for thousands of other flights first.

Two methods are used for specifying where to start reading. In the first one, every read operation gives the position in the file to start reading at. In the second one, a special operation, seek, is provided to set the current position. After a seek, the file can be read sequentially from the now-current position.

In some older mainframe operating systems, files are classified as being either sequential or random access at the time they are created. This allows the system to use different storage techniques for the two classes. Modern operating systems do not make this distinction. All their files are automatically random access.

6.1.5 File Attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was created and the file's size. We will call these extra items the file's **attributes**. The list of attributes varies considerably from system to system. The table of Fig. 6-4 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system.

The first four attributes relate to the file's protection and tell who may access it and who may not. All kinds of schemes are possible, some of which we will study later. In some systems the user must present a password to access a file, in which case the password must be one of the attributes.

The flags are bits or short fields that control or enable some specific property. Hidden files, for example, do not appear in listings of all the files. The archive flag is a bit that keeps track of whether the file has been backed up. The backup program clears it, and the operating system sets it whenever a file is changed. In this way, the backup program can tell which files need backing up. The temporary flag allows a file to be marked for automatic deletion when the process that created it terminates.

The record length, key position, and key length fields are only present in files whose records can be looked up using a key. They provide the information required to find the keys.

The various times keep track of when the file was created, most recently accessed and most recently modified. These are useful for a variety of purposes. For example, a source file that has been modified after the creation of the corresponding object file needs to be recompiled. These fields provide the necessary information.

The current size tells how big the file is at present. Some old mainframe operating systems require the maximum size to be specified when the file is created, in order to let the operating system reserve the maximum amount of storage in ad-

Spotify USA Inc. Exhibit 1009 Page 98

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure 6-4. Some possible file attributes.

vance. Workstation and personal computer operating systems are clever enough to do without this feature.

6.1.6 File Operations

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. Below is a discussion of the most common system calls relating to files.

1. Create. The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.
2. Delete. When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.
3. Open. Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.

Spotify USA Inc. Exhibit 1009 Page 99

4. Close. When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet.

5. Read. Data are read from file. Usually, the bytes come from the current position. The caller must specify how much data are needed and must also provide a buffer to put them in.

6. Write. Data are written to the file, again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.

7. Append. This call is a restricted form of write. It can only add data to the end of the file. Systems that provide a minimal set of system calls do not generally have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.

8. Seek. For random access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.

9. Get attributes. Processes often need to read file attributes to do their work. For example, the UNIX *make* program is commonly used to manage software development projects consisting of many source files. When *make* is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

10. Set attributes. Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection mode information is an obvious example. Most of the flags also fall in this category.

11. Rename. It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

Spotify USA Inc. Exhibit 1009 Page 100

6.1.7 An Example Program Using File System Calls

In this section we will examine a simple UNIX program that copies one file from its source file to a destination file. It is listed in Fig. 6-5. The program has minimal functionality and even worse error reporting, but it gives a reasonable idea of how some of the system calls related to files work.

The program, *copyfile*, can be called, for example, by the command line

```
copyfile abc xyz
```

to copy the file *abc* to *xyz*. If *xyz* already exists, it will be overwritten. Otherwise, it will be created. The program must be called with exactly two arguments, both legal file names.

The four *#include* statements near the top of the program cause a large number of definitions and function prototypes to be included in the program. These are needed to make the program conformant to the relevant international standards, but will not concern us further. The next line is a function prototype for *main*, something required by ANSI C, but also not important for our purposes.

The first *#define* statement is a macro definition that defines the string *BUF_SIZE* as a macro that expands into the number 4096. The program will read and write in chunks of 4096 bytes. It is considered good programming practice to give names to constants like this and to use the names instead of the constants. Not only does this convention make programs easier to read, but it also makes them easier to maintain. The second *#define* statement determines who can access the output file.

The main program is called *main*, and it has two arguments, *argc*, and *argv*. These are supplied by the operating system when the program is called. The first one tells how many strings were present on the command line that invoked the program, including the program name. It should be 3. The second one is an array of pointers to the arguments. In the example call given above, the elements of this array would contain pointers to the following values:

```
argv[0] = "copyfile"
argv[1] = "abc"
argv[2] = "xyz"
```

It is via this array that the program accesses its arguments.

Five variables are declared. The first two, *in_fd* and *out_fd*, will hold the **file descriptors**, small integers returned when a file is opened. The next two, *rd_count* and *wt_count*, are the byte counts returned by the read and write system calls, respectively. The last one, *buffer*, is the buffer used to hold the data read and supply the data to be written.

The first actual statement checks *argc* to see if it is 3. If not, it exits with status code 1. Any status code other than 0 means that an error has occurred. The

Spotify USA Inc. Exhibit 1009 Page 101

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);             /* syntax error if argc is not 3 */

    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY);    /* open the source file */
    if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
    if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
    if (rd_count <= 0) break;           /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                  /* no error on last read */
        exit(0);
    else
        exit(5);                        /* error on last read */
}
```

Figure 6-5. A simple program to copy a file.

Spotify USA Inc. Exhibit 1009 Page 102

status code is the only error reporting present in this program. A production version would normally print error messages as well.

Then we try to open the source file and create the destination file. If the source file is successfully opened, the system assigns a small integer to *in_fd*, to identify the file. Subsequent calls must include this integer so the system knows which file it wants. Similarly, if the destination is successfully created, *out_fd* is given a value to identify it. The second argument to *creat* sets the protection mode. If either the open or the create fails, the corresponding file descriptor is set to −1, and the program exits with an error code.

Now comes the copy loop. It starts by trying to read in 4 KB of data to *buffer*. It does this by calling the library procedure *read*, which actually invokes the read system call. The first parameter identifies the file, the second gives the buffer, and the third tells how many bytes to read. The value assigned to *rd_count* gives the number of bytes actually read. Normally, this will be 4096, except if fewer bytes are remaining in the file. When end of file is reached, it will be 0. If *rd_count* is ever zero or negative, the copying cannot continue, so the *break* statement is executed to terminate the (otherwise endless) loop.

The call to *write* outputs the buffer to the destination file. The first parameter identifies the file, the second gives the buffer, and the third tells how many bytes to write, analogous to *read*. Note that the byte count is the number of bytes actually read, not *BUF_SIZE*. This point is important because the last read will not return 4096, unless the file just happens to be a multiple of 4 KB.

When the entire file has been processed, the first call beyond the end of file will return 0 to *rd_count*, which will make it exit the loop. At this point the two files are closed and the program exits with a status indicating normal termination.

Although the Windows system calls are different from those of UNIX, the general structure of a command-line Windows program to copy a file is moderately similar to that of Fig. 6-5. We will examine the Windows 2000 calls in Chap. 11.

6.1.8 Memory-Mapped Files

Many programmers feel that accessing files as shown above is cumbersome and inconvenient, especially when compared to accessing ordinary memory. For this reason, some operating systems, starting with MULTICS, have provided a way to map files into the address space of a running process. Conceptually, we can imagine the existence of two new system calls, map and unmap. The former gives a file name and a virtual address, which causes the operating system to map the file into the address space at the virtual address.

For example, suppose that a file, *f*, of length 64 KB, is mapped into the virtual address starting at address 512K. Then any machine instruction that reads the contents of the byte at 512K gets byte 0 of the file, and so on. Similarly, a write to address 512K + 21000 modifies byte 21000 of the file. When the process

Spotify USA Inc. Exhibit 1009 Page 103

terminates, the modified file is left on the disk, just as though it had been changed by a combination of seek and write system calls.

What actually happens is that the system's internal tables are changed to make the file become the backing store for the memory region 512K to 576K. Thus a read from 512K causes a page fault, bringing in page 0 of the file. Similarly, a write to 512K + 1100 causes a page fault, bringing in the page containing that address, after which the write to memory can take place. If that page is ever evicted by the page replacement algorithm, it is written back to the appropriate place in the file. When the process finishes, all mapped, modified pages are written back to their files.

File mapping works best in a system that supports segmentation. In such a system, each file can be mapped onto its own segment so that byte *k* in the file is also byte *k* in the segment. In Fig. 6-6(a) we see a process that has two segments, text and data. Suppose that this process copies files, like the program of Fig. 6-5. First it maps the source file, say, *abc*, onto a segment. Then it creates an empty segment and maps it onto the destination file, *xyz* in our example. These operations give the situation shown in Fig. 6-6(b).

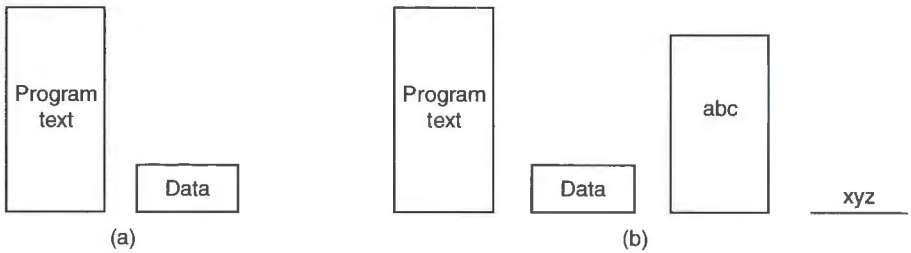


Figure 6-6. (a) A segmented process before mapping files into its address space. (b) The process after mapping an existing file *abc* into one segment and creating a new segment for file *xyz*.

At this point the process can copy the source segment into the destination segment using an ordinary copy loop. No read or write system calls are needed. When it is all done, it can execute the unmap system call to remove the files from the address space and then exit. The output file, *xyz*, will now exist, as though it had been created in the conventional way.

Although file mapping eliminates the need for I/O and thus makes programming easier, it introduces a few problems of its own. First, it is hard for the system to know the exact length of the output file, *xyz*, in our example. It can easily tell the number of the highest page written, but it has no way of knowing how many bytes in that page were written. Suppose that the program only uses page 0, and after execution all the bytes are still 0 (their initial value). Maybe *xyz* is a file consisting of 10 zeros. Maybe it is a file consisting of 100 zeros. Maybe it is a file consisting of 1000 zeros. Who knows? The operating system cannot tell. All it can do is create a file whose length is equal to the page size.

Spotify USA Inc. Exhibit 1009 Page 104

A second problem can (potentially) occur if a file is mapped in by one process and opened for conventional reading by another. If the first process modifies a page, that change will not be reflected in the file on disk until the page is evicted. The system has to take great care to make sure the two processes do not see inconsistent versions of the file.

A third problem with mapping is that a file may be larger than a segment, or even larger than the entire virtual address space. The only way out is to arrange the map system call to be able to map a portion of a file, rather than the entire file. Although this works, it is clearly less satisfactory than mapping the entire file.

6.2 DIRECTORIES

To keep track of files, file systems normally have **directories** or **folders**, which, in many systems, are themselves files. In this section we will discuss directories, their organization, their properties, and the operations that can be performed on them.

6.2.1 Single-Level Directory Systems

The simplest form of directory system is having one directory containing all the files. Sometimes it is called the **root directory**, but since it is the only one, the name does not matter much. On early personal computers, this system was common, in part because there was only one user. Interestingly enough, the world's first supercomputer, the CDC 6600, also had only a single directory for all files, even though it was used by many users at once. This decision was no doubt made to keep the software design simple.

An example of a system with one directory is given in Fig. 6-7. Here the directory contains four files. The file *owners* are shown in the figure, not the file *names* (because the owners are important to the point we are about to make). The advantages of this scheme are its simplicity and the ability to locate files quickly—there is only one place to look, after all.

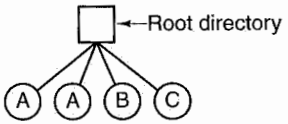


Figure 6-7. A single-level directory system containing four files, owned by three different people, *A*, *B*, and *C*.

The problem with having only one directory in a system with multiple users is that different users may accidentally use the same names for their files. For example, if user *A* creates a file called *mailbox*, and then later user *B* also creates a file called *mailbox*, *B*'s file will overwrite *A*'s file. Consequently, this scheme is

Spotify USA Inc. Exhibit 1009 Page 105

not used on multiuser systems any more, but could be used on a small embedded system, for example, a system in a car that was designed to store user profiles for a small number of drivers.

6.2.2 Two-level Directory Systems

To avoid conflicts caused by different users choosing the same file name for their own files, the next step up is giving each user a private directory. In that way, names chosen by one user do not interfere with names chosen by a different user and there is no problem caused by the same name occurring in two or more directories. This design leads to the system of Fig. 6-8. This design could be used, for example, on a multiuser computer or on a simple network of personal computers that shared a common file server over a local area network.

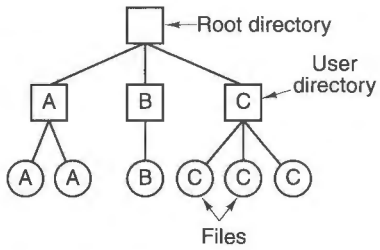


Figure 6-8. A two-level directory system. The letters indicate the owners of the directories and files.

Implicit in this design is that when a user tries to open a file, the system knows which user it is in order to know which directory to search. As a consequence, some kind of login procedure is needed, in which the user specifies a login name or identification, something not required with a single-level directory system.

When this system is implemented in its most basic form, users can only access files in their own directories. However, a slight extension is to allow users to access other users' files by providing some indication of whose file is to be opened. Thus, for example,

```
open("x")
```

might be the call to open a file called *x* in the user's directory, and

```
open("nancy/x")
```

might be the call to open a file *x* in the directory of another user, Nancy.

One situation in which users need to access files other than their own is to execute system binary programs. Having copies of all the utility programs present in each directory clearly is inefficient. At the very least, there is a need for a system directory with the executable binary programs.

Spotify USA Inc. Exhibit 1009 Page 106

6.2.3 Hierarchical Directory Systems

The two-level hierarchy eliminates name conflicts among users but is not satisfactory for users with a large number of files. Even on a single-user personal computer, it is inconvenient. It is quite common for users to want to group their files together in logical ways. A professor, for example, might have a collection of files that together form a book that he is writing for one course, a second collection of files containing student programs submitted for another course, a third group of files containing the code of an advanced compiler-writing system he is building, a fourth group of files containing grant proposals, as well as other files for electronic mail, minutes of meetings, papers he is writing, games, and so on. Some way is needed to group these files together in flexible ways chosen by the user.

What is needed is a general hierarchy (i.e., a tree of directories). With this approach, each user can have as many directories as are needed so that files can be grouped together in natural ways. This approach is shown in Fig. 6-9. Here, the directories *A*, *B*, and *C* contained in the root directory each belong to a different user, two of whom have created subdirectories for projects they are working on.

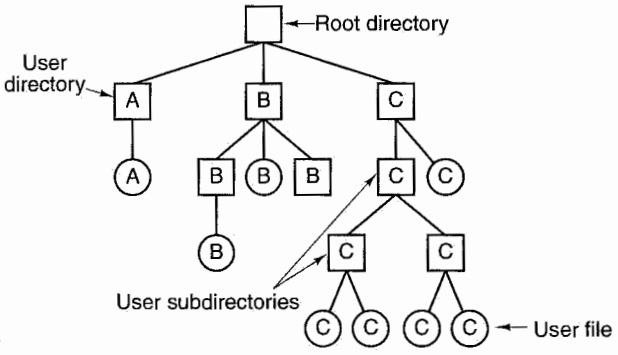


Figure 6-9. A hierarchical directory system.

The ability for users to create an arbitrary number of subdirectories provides a powerful structuring tool for users to organize their work. For this reason, nearly all modern file systems are organized in this manner.

6.2.4 Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used. In the first method, each file is given an **absolute path name** consisting of the path from the root directory to the file. As an example, the path */usr/ast/mailbox* means that the

Spotify USA Inc. Exhibit 1009 Page 107

root directory contains a subdirectory *usr*, which in turn contains a subdirectory *ast*, which contains the file *mailbox*. Absolute path names always start at the root directory and are unique. In UNIX the components of the path are separated by /. In Windows the separator is \. In MULTICS it was >. Thus the same path name would be written as follows in these three systems:

Windows	\usr\ast\mailbox
UNIX	/usr/ast/mailbox
MULTICS	>usr>ast>mailbox

No matter which character is used, if the first character of the path name is the separator, then the path is absolute.

The other kind of name is the **relative path name**. This is used in conjunction with the concept of the **working directory** (also called the **current directory**). A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory. For example, if the current working directory is */usr/ast*, then the file whose absolute path is */usr/ast/mailbox* can be referenced simply as *mailbox*. In other words, the UNIX command

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
```

and the command

```
cp mailbox mailbox.bak
```

do exactly the same thing if the working directory is */usr/ast*. The relative form is often more convenient, but it does the same thing as the absolute form.

Some programs need to access a specific file without regard to what the working directory is. In that case, they should always use absolute path names. For example, a spelling checker might need to read */usr/lib/dictionary* to do its work. It should use the full, absolute path name in this case because it does not know what the working directory will be when it is called. The absolute path name will always work, no matter what the working directory is.

Of course, if the spelling checker needs a large number of files from */usr/lib*, an alternative approach is for it to issue a system call to change its working directory to */usr/lib*, and then use just *dictionary* as the first parameter to open. By explicitly changing the working directory, it knows for sure where it is in the directory tree, so it can then use relative paths.

Each process has its own working directory, so when a process changes its working directory and later exits, no other processes are affected and no traces of the change are left behind in the file system. In this way it is always perfectly safe for a process to change its working directory whenever that is convenient. On the other hand, if a *library procedure* changes the working directory and does not change back to where it was when it is finished, the rest of the program may not work since its assumption about where it is may now suddenly be invalid. For

Spotify USA Inc. Exhibit 1009 Page 108

this reason, library procedures rarely change the working directory, and when they must, they always change it back again before returning.

Most operating systems that support a hierarchical directory system have two special entries in every directory, "." and "..", generally pronounced "dot" and "dotdot." Dot refers to the current directory; dotdot refers to its parent. To see how these are used, consider the UNIX file tree of Fig. 6-10. A certain process has */usr/ast* as its working directory. It can use .. to go up the tree. For example, it can copy the file */usr/lib/dictionary* to its own directory using the command

```
cp ../lib/dictionary .
```

The first path instructs the system to go upward (to the *usr* directory), then to go down to the directory *lib* to find the file *dictionary*.

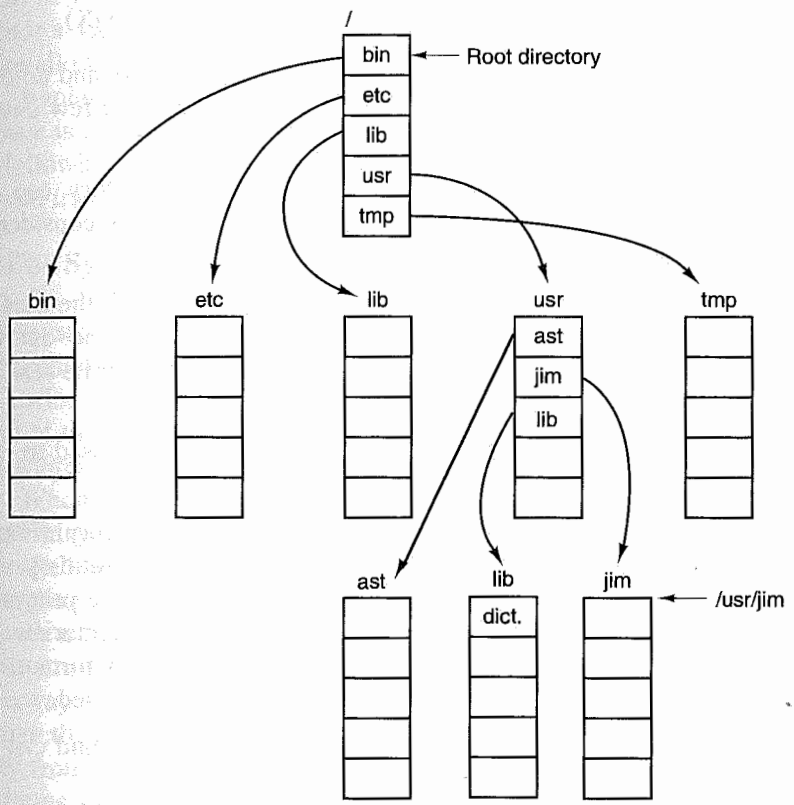


Figure 6-10. A UNIX directory tree.

The second argument (dot) names the current directory. When the *cp* command gets a directory name (including dot) as its last argument, it copies all the files there. Of course, a more normal way to do the copy would be to type

```
cp /usr/lib/dictionary .
```

Spotify USA Inc. Exhibit 1009 Page 109

Here the use of dot saves the user the trouble of typing *dictionary* a second time. Nevertheless, typing

```
cp /usr/lib/dictionary dictionary
```

also works fine, as does

```
cp /usr/lib/dictionary /usr/ast/dictionary
```

All of these do exactly the same thing.

6.2.5 Directory Operations

The allowed system calls for managing directories exhibit more variation from system to system than system calls for files. To give an impression of what they are and how they work, we will give a sample (taken from UNIX).

1. Create. A directory is created. It is empty except for dot and dotdot, which are put there automatically by the system (or in a few cases, by the *mkdir* program).
2. Delete. A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered empty as these cannot usually be deleted.
3. Opendir. Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.
4. Closedir. When a directory has been read, it should be closed to free up internal table space.
5. Readdir. This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structures is being used.
6. Rename. In many respects, directories are just like files and can be renamed the same way files can be.
7. Link. Linking is a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories. A link of this kind, which increments the counter in the file's i-node (to keep track of the number of directory entries containing the file), is sometimes called a **hard link**.

Spotify USA Inc. Exhibit 1009 Page 110

8. Unlink. A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. The others remain. In UNIX, the system call for deleting files (discussed earlier) is, in fact, unlink.

The above list gives the most important calls, but there are a few others as well, for example, for managing the protection information associated with a directory.

6.3 FILE SYSTEM IMPLEMENTATION

Now it is time to turn from the user's view of the file system to the implementor's view. Users are concerned with how files are named, what operations are allowed on them, what the directory tree looks like, and similar interface issues. Implementors are interested in how files and directories are stored, how disk space is managed, and how to make everything work efficiently and reliably. In the following sections we will examine a number of these areas to see what the issues and trade-offs are.

6.3.1 File System Layout

File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition. Sector 0 of the disk is called the **MBR** (**Master Boot Record**) and is used to boot the computer. The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition. One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR. The first thing the MBR program does is locate the active partition, read in its first block, called the **boot block**, and execute it. The program in the boot block loads the operating system contained in that partition. For uniformity, every partition starts with a boot block, even if it does not contain a bootable operating system. Besides, it might contain one in the future, so reserving a boot block is a good idea anyway.

Other than starting with a boot block, the layout of a disk partition varies strongly from file system to file system. Often the file system will contain some of the items shown in Fig. 6-11. The first one is the **superblock**. It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched. Typical information in the superblock includes a magic number to identify the file system type, the number of blocks in the file system, and other key administrative information.

Next might come information about free blocks in the file system, for example in the form of a bitmap or a list of pointers. This might be followed by the i-

Spotify USA Inc. Exhibit 1009 Page 111

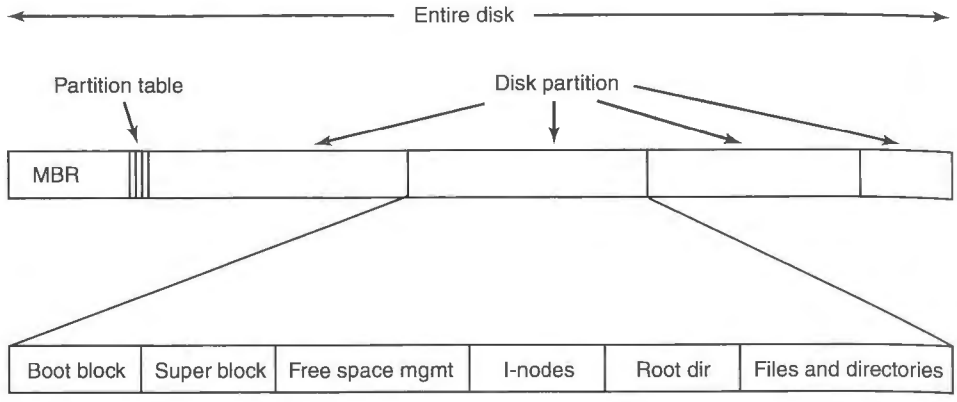


Figure 6-11. A possible file system layout.

nodes, an array of data structures, one per file, telling all about the file. After that might come the root directory, which contains the top of the file system tree. Finally, the remainder of the disk typically contains all the other directories and files.

6.3.2 Implementing Files

Probably the most important issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different operating systems. In this section, we will examine a few of them.

Contiguous Allocation

The simplest allocation scheme is to store each file as a contiguous run of disk blocks. Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. With 2-KB blocks, it would be allocated 25 consecutive blocks.

We see an example of contiguous storage allocation in Fig. 6-12(a). Here the first 40 disk blocks are shown, starting with block 0 on the left. Initially, the disk was empty. Then a file *A*, of length four blocks was written to disk starting at the beginning (block 0). After that a six-block file, *B*, was written starting right after the end of file *A*. Note that each file begins at the start of a new block, so that if file *A* was really 3½ blocks, some space is wasted at the end of the last block. In the figure, a total of seven files are shown, each one starting at the block following the end of the previous one. Shading is used just to make it easier to tell the files apart.

Contiguous disk space allocation has two significant advantages. First, it is simple to implement because keeping track of where a file's blocks are is reduced

Spotify USA Inc. Exhibit 1009 Page 112

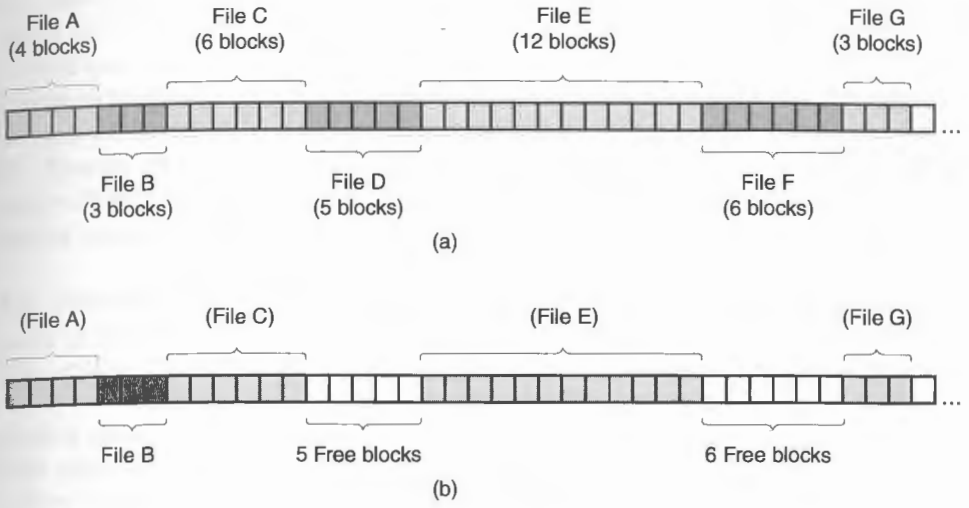


Figure 6-12. (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed.

to remembering two numbers: the disk address of the first block and the number of blocks in the file. Given the number of the first block, the number of any other block can be found by a simple addition.

Second, the read performance is excellent because the entire file can be read from the disk in a single operation. Only one seek is needed (to the first block). After that, no more seeks or rotational delays are needed so data come in at the full bandwidth of the disk. Thus contiguous allocation is simple to implement and has high performance.

Unfortunately, contiguous allocation also has a significant drawback: in time, the disk becomes fragmented. To see how this comes about, examine Fig. 6-12(b). Here two files, *D* and F have been removed. When a file is removed, its blocks are freed, leaving a run of free blocks on the disk. The disk is not compacted on the spot to squeeze out the hole since that would involve copying all the blocks following the hole, potentially millions of blocks. As a result, the disk ultimately consists of files and holes, as illustrated in the figure.

Initially, this fragmentation is not a problem since each new file can be written at the end of disk, following the previous one. However, eventually the disk will fill up and it will become necessary to either compact the disk, which is prohibitively expensive, or to reuse the free space in the holes. Reusing the space requires maintaining a list of holes, which is doable. However, when a new file is to be created, it is necessary to know its final size in order to choose a hole of the correct size to place it in.

Imagine the consequences of such a design. The user starts a text editor or word processor in order to type a document. The first thing the program asks is

Spotify USA Inc. Exhibit 1009 Page 113

how many bytes the final document will be. The question must be answered or the program will not continue. If the number given ultimately proves too small, the program has to terminate prematurely because the disk hole is full and there is no place to put the rest of the file. If the user tries to avoid this problem by giving an unrealistically large number as the final size, say, 100 MB, the editor may be unable to find such a large hole and announce that the file cannot be created. Of course, the user would be free to start the program again and say 50 MB this time, and so on until a suitable hole was located. Still, this scheme is not likely to lead to happy users.

However, there is one situation in which contiguous allocation is feasible and, in fact, widely used: on CD-ROMs. Here all the file sizes are known in advance and will never change during subsequent use of the CD-ROM file system. We will study the most common CD-ROM file system later in this chapter.

As we mentioned in Chap. 1, history often repeats itself in computer science as new generations of technology occur. Contiguous allocation was actually used on magnetic disk file systems years ago due to its simplicity and high performance (user friendliness did not count for much then). Then the idea was dropped due to the nuisance of having to specify final file size at file creation time. But with the advent of CD-ROMs, DVDs, and other write-once optical media, suddenly contiguous files are a good idea again. It is thus important to study old systems and ideas that were conceptually clean and simple because they may be applicable to future systems in surprising ways.

Linked List Allocation

The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Fig. 6-13. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation (except for internal fragmentation in the last block). Also, it is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there.

On the other hand, although reading a file sequentially is straightforward, random access is extremely slow. To get to block n, the operating system has to start at the beginning and read the $n - 1$ blocks prior to it, one at a time. Clearly, doing so many reads will be painfully slow.

Also, the amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes. While not fatal, having a peculiar size is less efficient because many programs read and write in blocks whose size is a power of two. With the first few bytes of each block occupied to a pointer to the next block, reads of the full block size require acquiring and concatenating information from two disk blocks, which generates extra overhead due to the copying.

Spotify USA Inc. Exhibit 1009 Page 114

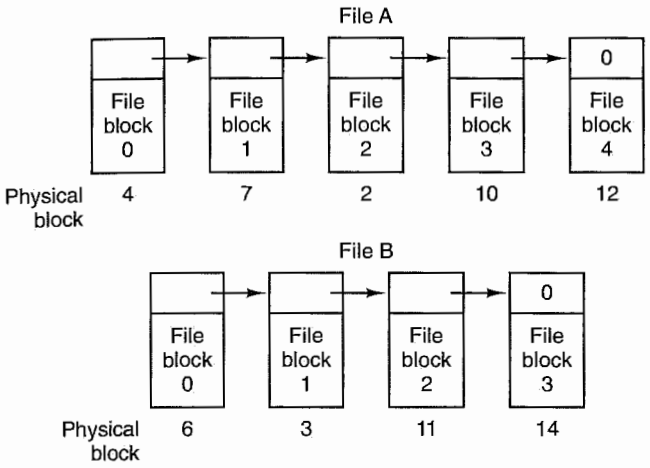


Figure 6-13. Storing a file as a linked list of disk blocks.

Linked List Allocation Using a Table in Memory

Both disadvantages of the linked list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table in memory. Figure 6-14 shows what the table looks like for the example of Fig. 6-13. In both figures, we have two files. File *A* uses disk blocks 4, 7, 2, 10, and 12, in that order, and file *B* uses disk blocks 6, 3, 11, and 14, in that order. Using the table of Fig. 6-14, we can start with block 4 and follow the chain all the way to the end. The same can be done starting with block 6. Both chains are terminated with a special marker (e.g., −1) that is not a valid block number. Such a table in main memory is called a **FAT (File Allocation Table)**.

Using this organization, the entire block is available for data. Furthermore, random access is much easier. Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory, so it can be followed without making any disk references. Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is.

The primary disadvantage of this method is that the entire table must be in memory all the time to make it work. With a 20-GB disk and a 1-KB block size, the table needs 20 million entries, one for each of the 20 million disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 60 MB or 80 MB of main memory all the time, depending on whether the system is optimized for space or time. Conceivably the table could be put in pageable memory, but it would still occupy a great deal of virtual memory and disk space as well as generating extra paging traffic.

Spotify USA Inc. Exhibit 1009 Page 115

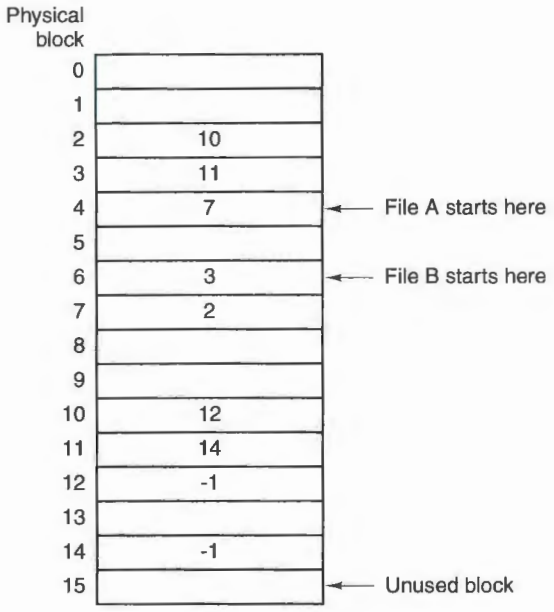


Figure 6-14. Linked list allocation using a file allocation table in main memory.

I-nodes

Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an **i-node** (**index-node**), which lists the attributes and disk addresses of the file's blocks. A simple example is depicted in Fig. 6-15. Given the i-node, it is then possible to find all the blocks of the file. The big advantage of this scheme over linked files using an in-memory table is that the i-node need only be in memory when the corresponding file is open. If each i-node occupies n bytes and a maximum of k files may be open at once, the total memory occupied by the array holding the i-nodes for the open files is only kn bytes. Only this much space need be reserved in advance.

This array is usually far smaller than the space occupied by the file table described in the previous section. The reason is simple. The table for holding the linked list of all disk blocks is proportional in size to the disk itself. If the disk has n blocks, the table needs n entries. As disks grow larger, this table grows linearly with them. In contrast, the i-node scheme requires an array in memory whose size is proportional to the maximum number of files that may be open at once. It does not matter if the disk is 1 GB or 10 GB or 100 GB.

One problem with i-nodes is that if each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit? One solution is to reserve the last disk address not for a data block, but instead for the address

Spotify USA Inc. Exhibit 1009 Page 116

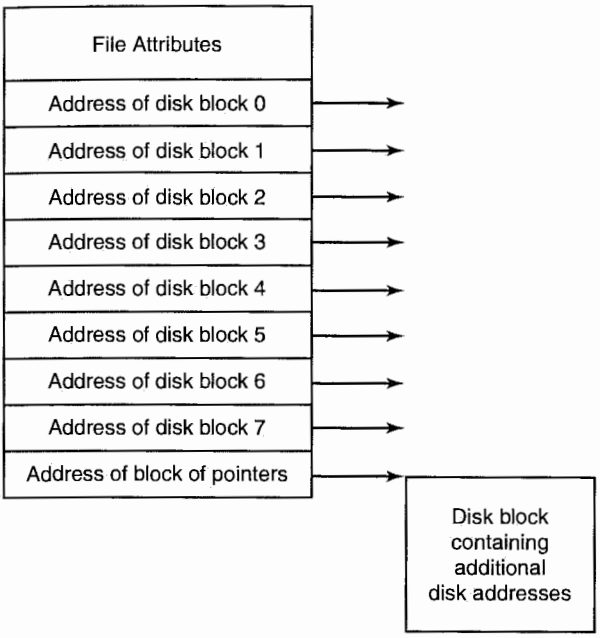


Figure 6-15. An example i-node.

of a block containing more disk block addresses, as shown in Fig. 6-15. Even more advanced would be two or more such blocks containing disk addresses or even disk blocks pointing to other disk blocks full of addresses. We will come back to i-nodes when studying UNIX later.

6.3.3 Implementing Directories

Before a file can be read, it must be opened. When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry. The directory entry provides the information needed to find the disk blocks. Depending on the system, this information may be the disk address of the entire file (contiguous allocation), the number of the first block (both linked list schemes), or the number of the i-node. In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data.

A closely related issue is where the attributes should be stored. Every file system maintains file attributes, such as each file's owner and creation time, and they must be stored somewhere. One obvious possibility is to store them directly in the directory entry. Many systems do precisely that. This option is shown in Fig. 6-16(a) In this simple design, a directory consists of a list of fixed-size

entries, one per file, containing a (fixed-length) file name, a structure of the file attributes, and one or more disk addresses (up to some maximum) telling where the disk blocks are.

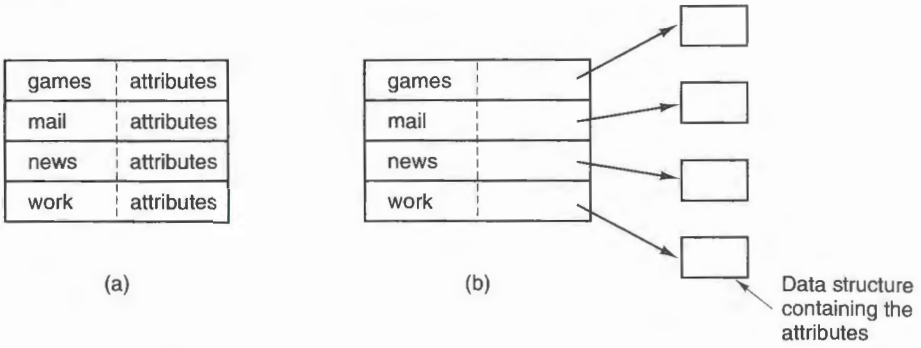


Figure 6-16. (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

For systems that use i-nodes, another possibility for storing the attributes is in the i-nodes, rather than in the directory entries. In that case, the directory entry can be shorter: just a file name and an i-node number. This approach is illustrated in Fig. 6-16(b). As we shall see later, this method has certain advantages over putting them in the directory entry. The two approaches shown in Fig. 6-16 correspond to MS-DOS/Windows and UNIX, respectively, as we will see later in this chapter.

So far we have assumed that files have short, fixed-length names. In MS-DOS files have a 1-8 character base name and an optional extension of 1-3 characters. In UNIX Version 7, file names were 1-14 characters, including any extensions. However, nearly all modern operating systems support longer, variable-length file names. How can these be implemented?

The simplest approach is to set a limit on file name length, typically 255 characters, and then use one of the designs of Fig. 6-16 with 255 characters reserved for each file name. This approach is simple, but wastes a great deal of directory space, since few files have such long names. For efficiency reasons, a different structure is desirable.

One alternative is to give up the idea that all directory entries are the same size. With this method, each directory entry contains a fixed portion, typically starting with the length of the entry, and then followed by data with a fixed format, usually including the owner, creation time, protection information, and other attributes. This fixed-length header is followed by the actual file name, however long it may be, as shown in Fig. 6-17(a) in big-endian format (e.g., SPARC). In this example we have three files, *project-budget*, *personnel*, and *foo*. Each file name is terminated by a special character (usually 0), which is represented in the

Spotify USA Inc. Exhibit 1009 Page 118

figure by a box with a cross in it. To allow each directory entry to begin on a word boundary, each file name is filled out to an integral number of words, shown by shaded boxes in the figure.

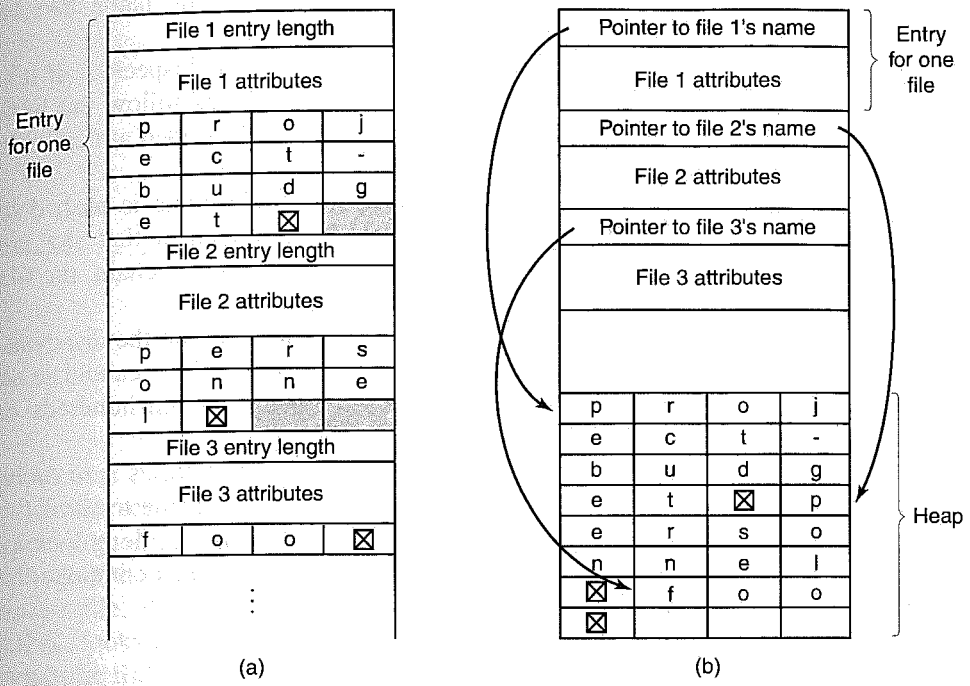


Figure 6-17. Two ways of handling long file names in a directory. (a) In-line. (b) In a heap.

A disadvantage of this method is that when a file is removed, a variable-sized gap is introduced into the directory into which the next file to be entered may not fit. This problem is the same one we saw with contiguous disk files, only now compacting the directory is feasible because it is entirely in memory. Another problem is that a single directory entry may span multiple pages, so a page fault may occur while reading a file name.

Another way to handle variable-length names is to make the directory entries themselves all fixed length and keep the file names together in a heap at the end of the directory, as shown in Fig. 6-17(b). This method has the advantage that when an entry is removed, the next file entered will always fit there. Of course, the heap must be managed and page faults can still occur while processing file names. One minor win here is that there is no longer any real need for file names to begin at word boundaries, so no filler characters are needed after file names in Fig. 6-17(b) and they are in Fig. 6-17(a).

In all of the designs so far, directories are searched linearly from beginning to end when a file name has to be looked up. For extremely long directories, linear

Spotify USA Inc. Exhibit 1009 Page 119

searching can be slow. One way to speed up the search is to use a hash table in each directory. Call the size of the table n. To enter a file name, the name is hashed onto a value between 0 and $n - 1$, for example, by dividing it by n and taking the remainder. Alternatively, the words comprising the file name can be added up and this quantity divided by n, or something similar.

Either way, the table entry corresponding to the hash code is inspected. If it is unused, a pointer is placed there to the file entry. File entries follow the hash table. If that slot is already in use, a linked list is constructed, headed at that table entry and threading through all entries with the same hash value.

Looking up a file follows the same procedure. The file name is hashed to select a hash table entry. All the entries on the chain headed at that slot are checked to see if the file name is present. If the name is not on the chain, the file is not present in the directory.

Using a hash table has the advantage of much faster lookup, but the disadvantage of a more complex administration. It is only really a serious candidate in systems where it is expected that directories will routinely contain hundreds or thousands of files.

A completely different way to speed up searching large directories is to cache the results of searches. Before starting a search, a check is first made to see if the file name is in the cache. If so, it can be located quickly, avoiding a long lookup. Of course, caching only works if a relatively small number of files comprise the majority of the lookups.

6.3.4 Shared Files

When several users are working together on a project, they often need to share files. As a result, it is often convenient for a shared file to appear simultaneously in different directories belonging to different users. Figure 6-18 shows the file system of Fig. 6-9 again, only with one of *C*'s files now present in one of *B*'s directories as well. The connection between *B*'s directory and the shared file is called a **link**. The file system itself is now a **Directed Acyclic Graph**, or **DAG**, rather than a tree.

Sharing files is convenient, but it also introduces some problems. To start with, if directories really do contain disk addresses, then a copy of the disk addresses will have to be made in *B*'s directory when the file is linked. If either *B* or *C* subsequently appends to the file, the new blocks will be listed only in the directory of the user doing the append. The changes will not be visible to the other user, thus defeating the purpose of sharing.

This problem can be solved in two ways. In the first solution, disk blocks are not listed in directories, but in a little data structure associated with the file itself. The directories would then point just to the little data structure. This is the approach used in UNIX (where the little data structure is the i-node).

Spotify USA Inc. Exhibit 1009 Page 120

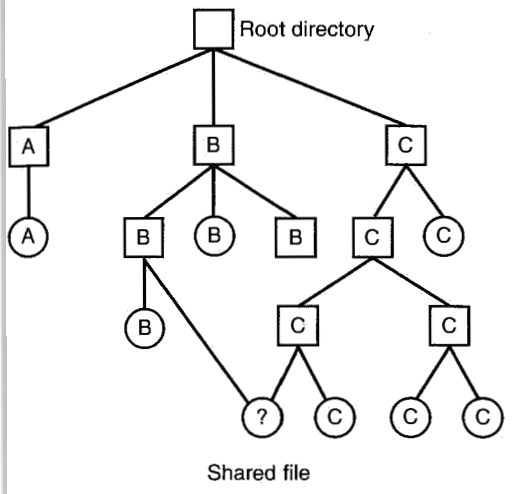


Figure 6-18. File system containing a shared file.

In the second solution, *B* links to one of *C*'s files by having the system create a new file, of type LINK, and entering that file in *B's* directory. The new file contains just the path name of the file to which it is linked. When *B* reads from the linked file, the operating system sees that the file being read from is of type LINK, looks up the name of the file, and reads that file. This approach is called **symbolic linking**.

Each of these methods has its drawbacks. In the first method, at the moment that *B* links to the shared file, the i-node records the file's owner as *C*. Creating a link does not change the ownership (see Fig. 6-19), but it does increase the link count in the i-node, so the system knows how many directory entries currently point to the file.

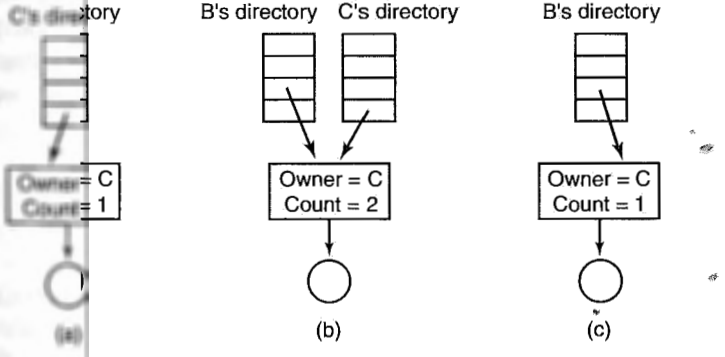


Figure 6-19. (a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

Spotify USA Inc. Exhibit 1009 Page 121

If *C* subsequently tries to remove the file, the system is faced with a problem. If it removes the file and clears the i-node, *B* will have a directory entry pointing to an invalid i-node. If the i-node is later reassigned to another file, *B*'s link will point to the wrong file. The system can see from the count in the i-node that the file is still in use, but there is no way for it to find all the directory entries for the file, in order to erase them. Pointers to the directories cannot be stored in the i-node because there can be an unlimited number of directories.

The only thing to do is remove *C*'s directory entry, but leave the i-node intact, with count set to 1, as shown in Fig. 6-19(c). We now have a situation in which *B* is the only user having a directory entry for a file owned by *C*. If the system does accounting or has quotas, *C* will continue to be billed for the file until *B* decides to remove it, if ever, at which time the count goes to 0 and the file is deleted.

With symbolic links this problem does not arise because only the true owner has a pointer to the i-node. Users who have linked to the file just have path names, not i-node pointers. When the *owner* removes the file, it is destroyed. Subsequent attempts to use the file via a symbolic link will fail when the system is unable to locate the file. Removing a symbolic link does not affect the file at all.

The problem with symbolic links is the extra overhead required. The file containing the path must be read, then the path must be parsed and followed, component by component, until the i-node is reached. All of this activity may require a considerable number of extra disk accesses. Furthermore, an extra i-node is needed for each symbolic link, as is an extra disk block to store the path, although if the path name is short, the system could store it in the i-node itself, as an optimization. Symbolic links have the advantage that they can be used to link to files on machines anywhere in the world, by simply providing the network address of the machine where the file resides in addition to its path on that machine.

There is also another problem introduced by links, symbolic or otherwise. When links are allowed, files can have two or more paths. Programs that start at a given directory and find all the files in that directory and its subdirectories will locate a linked file multiple times. For example, a program that dumps all the files in a directory and its subdirectories onto a tape may make multiple copies of a linked file. Furthermore, if the tape is then read into another machine, unless the dump program is clever, the linked file will be copied twice onto the disk, instead of being linked.

6.3.5 Disk Space Management

Files are normally stored on disk, so management of disk space is a major concern to file system designers. Two general strategies are possible for storing an *n* byte file: *n* consecutive bytes of disk space are allocated, or the file is split up into a number of (not necessarily) contiguous blocks. The same trade-off is present in memory management systems between pure segmentation and paging.

Spotify USA Inc. Exhibit 1009 Page 122

As we have seen, storing a file as a contiguous sequence of bytes has the obvious problem that if a file grows, it will probably have to be moved on the disk. The same problem holds for segments in memory, except that moving a segment in memory is a relatively fast operation compared to moving a file from one disk position to another. For this reason, nearly all file systems chop files up into fixed-size blocks that need not be adjacent.

Block Size

Once it has been decided to store files in fixed-size blocks, the question arises of how big the block should be. Given the way disks are organized, the sector, the track and the cylinder are obvious candidates for the unit of allocation (although these are all device dependent, which is a minus). In a paging system, the page size is also a major contender.

Having a large allocation unit, such as a cylinder, means that every file, even a 1-byte file, ties up an entire cylinder. Studies (Mullender and Tanenbaum, 1984) have shown that the median file size in UNIX environments is about 1 KB, so allocating a 32-KB block for each file would waste 31/32 or 97 percent of the total disk space.

On the other hand, using a small allocation unit means that each file will consist of many blocks. Reading each block normally requires a seek and a rotational delay, so reading a file consisting of many small blocks will be slow.

As an example, consider a disk with 131,072 bytes per track, a rotation time of 8.33 msec, and an average seek time of 10 msec. The time in milliseconds to read a block of k bytes is then the sum of the seek, rotational delay, and transfer times:

$$10 + 4.165 + (k/131072) \times 8.33$$

The solid curve of Fig. 6-20 shows the data rate for such a disk as a function of block size. To compute the space efficiency, we need to make an assumption about the mean file size. A recent measurement at the author's department, which has 1000 users and over 1 million UNIX disk files, gives a median size of 1680 bytes, meaning that half the files are smaller than 1680 bytes and half are larger. As an aside, the median is a better metric than the mean because a very small number of files can influence the mean enormously, but not the median. (The mean is in fact 10,845 bytes, due in part to a few 100 MB hardware manuals that happen to be online). For simplicity, let us assume all files are 2 KB, which leads to the dashed curve in Fig. 6-20 for the disk space efficiency.

The two curves can be understood as follows. The access time for a block is completely dominated by the seek time and rotational delay, so given that it is going to cost 14 msec to access a block, the more data that are fetched, the better. Hence the data rate goes up with block size (until the transfers take so long that

Spotify USA Inc. Exhibit 1009 Page 123

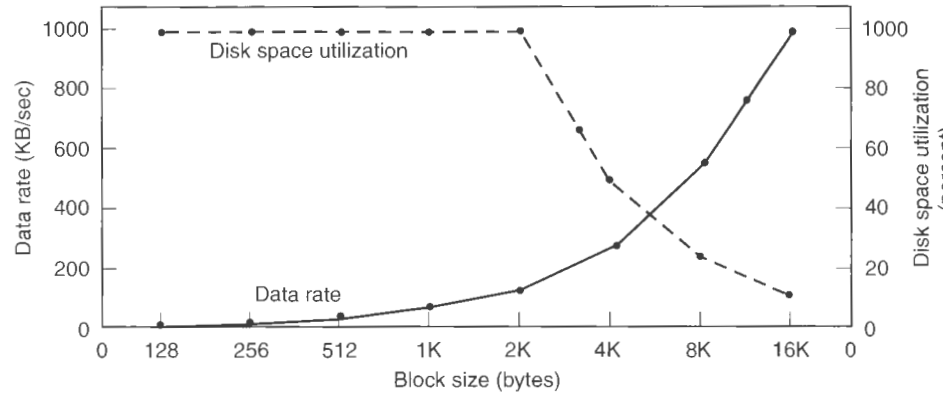


Figure 6-20. The solid curve (left-hand scale) gives the data rate of a disk. The dashed curve (right-hand scale) gives the disk space efficiency. All files are 2 KB.

the transfer time begins to dominate). With small blocks that are powers of two and 2-KB files, no space is wasted in a block. However, with 2-KB files and 4 KB or larger blocks, some disk space is wasted. In reality, few files are a multiple of the disk block size, so some space is always wasted in the last block of a file.

What the curves show, however, is that performance and space utilization are inherently in conflict. Small blocks are bad for performance but good for disk space utilization. A compromise size is needed. For this data, 4 KB might be a good choice, but some operating systems made their choices a long time ago, when the disk parameters and file sizes were different. For UNIX, 1 KB is commonly used. For MS-DOS the block size can be any power of two from 512 bytes to 32 KB, but is determined by the disk size and for reasons unrelated to these arguments (the maximum number of blocks on a disk partition is 2^{16}, which forces large blocks on large disks).

In an experiment to see if Windows NT file usage was appreciably different from UNIX file usage, Vogels made measurements on files at Cornell University (Vogels, 1999). He observed that NT file usage is more complicated than on UNIX. He wrote:

> *When we type a few characters in the* notepad *text editor, saving this to a file will trigger 26 system calls, including 3 failed open attempts, 1 file overwrite and 4 additional open and close sequences.*

Nevertheless, he observed a median size (weighted by usage) of files just read at 1 KB, files just written as 2.3 KB and files read and written as 4.2 KB. Given the fact that Cornell has much more large-scale scientific computing than the author's institution and the difference in measurement technique (static versus dynamic), the results are reasonably consistent with a median file size of around 2 KB.

Spotify USA Inc. Exhibit 1009 Page 124

Keeping Track of Free Blocks

Once a block size has been chosen, the next issue is how to keep track of free blocks. Two methods are widely used, as shown in Fig. 6-21. The first one consists of using a linked list of disk blocks, with each block holding as many free disk block numbers as will fit. With a 1-KB block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks. (One slot is needed for the pointer to the next block). A 16-GB disk needs a free list of maximum 16,794 blocks to hold all 2^{24} disk block numbers. Often free blocks are used to hold the free list.

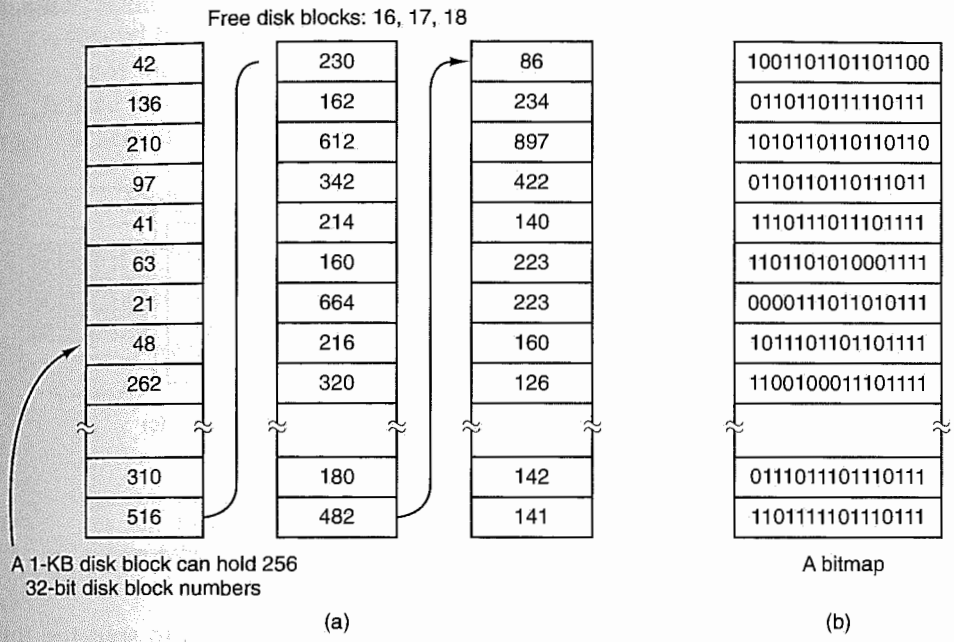


Figure 6-21. (a) Storing the free list on a linked list. (b) A bitmap.

The other free space management technique is the bitmap. A disk with n blocks requires a bitmap with n bits. Free blocks are represented by 1s in the map, allocated blocks by 0s (or vice versa). A 16-GB disk has 2^{24} 1-KB blocks and thus requires 2^{24} bits for the map, which requires 2048 blocks. It is not surprising that the bitmap requires less space, since it uses 1 bit per block, versus 32 bits in the linked list model. Only if the disk is nearly full (i.e., has few free blocks) will the linked list scheme require fewer blocks than the bitmap. On the other hand, if there are many blocks free, some of them can be borrowed to hold the free list without any loss of disk capacity.

When the free list method is used, only one block of pointers need be kept in main memory. When a file is created, the needed blocks are taken from the block

Spotify USA Inc. Exhibit 1009 Page 125

of pointers. When it runs out, a new block of pointers is read in from the disk. Similarly, when a file is deleted, its blocks are freed and added to the block of pointers in main memory. When this block fills up, it is written to disk.

Under certain circumstances, this method leads to unnecessary disk I/O. Consider the situation of Fig. 6-22(a), in which the block of pointers in memory has room for only two more entries. If a three-block file is freed, the pointer block overflows and has to be written to disk, leading to the situation of Fig. 6-22(b). If a three-block file is now written, the full block of pointers has to be read in again, taking us back to Fig. 6-22(a). If the three-block file just written was a temporary file, when it is freed, another disk write is needed to write the full block of pointers back to the disk. In short, when the block of pointers is almost empty, a series of short-lived temporary files can cause a lot of disk I/O.

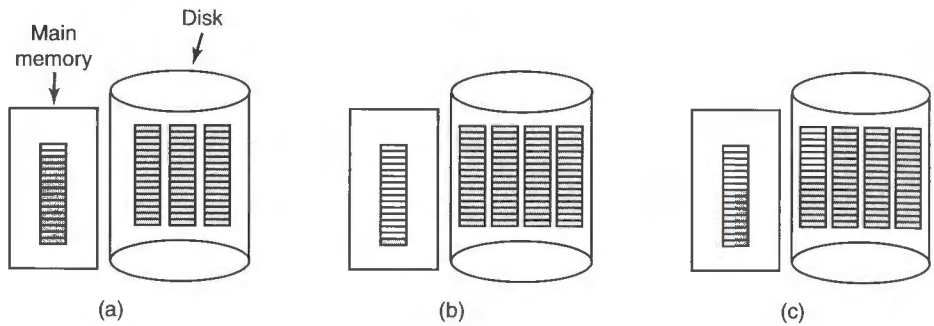


Figure 6-22. (a) An almost-full block of pointers to free disk blocks in memory and three blocks of pointers on disk. (b) Result of freeing a three-block file. (c) An alternative strategy for handling the three free blocks. The shaded entries represent pointers to free disk blocks.

An alternative approach that avoids most of this disk I/O is to split the full block of pointers. Thus instead of going from Fig. 6-22(a) to Fig. 6-22(b), we go from Fig. 6-22(a) to Fig. 6-22(c) when three blocks are freed. Now the system can handle a series of temporary files without doing any disk I/O. If the block in memory fills up, it is written to the disk and the half-full block from disk is read in. The idea here is to keep most of the pointer blocks on disk full (to minimize disk usage), but keep the one in memory about half full, so it can handle both file creation and file removal without disk I/O on the free list.

With a bitmap, it is also possible to keep just one block in memory, going to disk for another one only when it becomes full or empty. An additional benefit of this approach is that by doing all the allocation from a single block of the bitmap, the disk blocks will be close together, thus minimizing disk arm motion. Since the bitmap is a fixed-size data structure, if the kernel is (partially) paged, the bitmap can be put in virtual memory and have pages of it paged in as needed.

Spotify USA Inc. Exhibit 1009 Page 126

Disk Quotas

To prevent people from hogging too much disk space, multiuser operating systems often provide a mechanism for enforcing disk quotas. The idea is that the system administrator assigns each user a maximum allotment of files and blocks, and the operating system makes sure that the users do not exceed their quotas. A typical mechanism is described below.

When a user opens a file, the attributes and disk addresses are located and put into an open file table in main memory. Among the attributes is an entry telling who the owner is. Any increases in the file's size will be charged to the owner's quota.

A second table contains the quota record for every user with a currently open file, even if the file was opened by someone else. This table is shown in Fig. 6-23. It is an extract from a quota file on disk for the users whose files are currently open. When all the files are closed, the record is written back to the quota file.

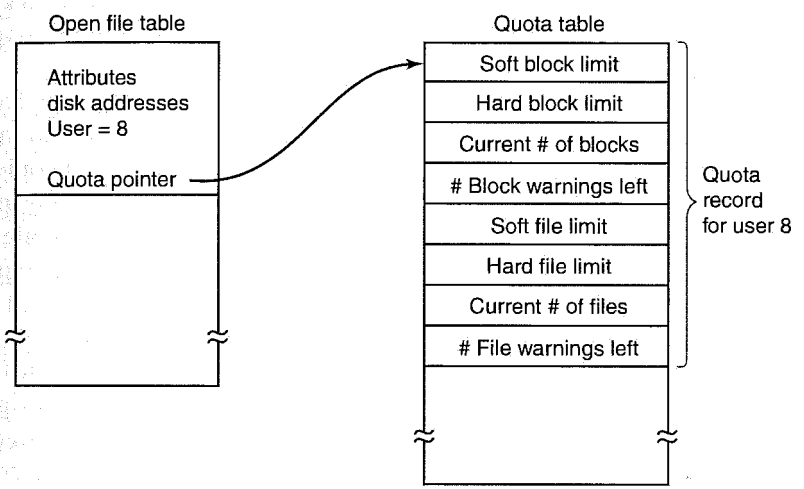


Figure 6-23. Quotas are kept track of on a per-user basis in a quota table.

When a new entry is made in the open file table, a pointer to the owner's quota record is entered into it, to make it easy to find the various limits. Every time a block is added to a file, the total number of blocks charged to the owner is incremented, and a check is made against both the hard and soft limits. The soft limit may be exceeded, but the hard limit may not. An attempt to append to a file when the hard block limit has been reached will result in an error. Analogous checks also exist for the number of files.

When a user attempts to log in, the system checks the quota file to see if the user has exceeded the soft limit for either number of files or number of disk blocks. If either limit has been violated, a warning is displayed, and the count of

warnings remaining is reduced by one. If the count ever gets to zero, the user has ignored the warning one time too many, and is not permitted to log in. Getting permission to log in again will require some discussion with the system administrator.

This method has the property that users may go above their soft limits during a login session, provided they remove the excess before logging out. The hard limits may never be exceeded.

6.3.6 File System Reliability

Destruction of a file system is often a far greater disaster than destruction of a computer. If a computer is destroyed by fire, lightning surges, or a cup of coffee poured onto the keyboard, it is annoying and will cost money, but generally a replacement can be purchased with a minimum of fuss. Inexpensive personal computers can even be replaced within an hour by just going to the dealer (except at universities, where issuing a purchase order takes three committees, five signatures, and 90 days).

If a computer's file system is irrevocably lost, whether due to hardware, software, or rats gnawing on the floppy disks, restoring all the information will be difficult, time consuming, and in many cases, impossible. For the people whose programs, documents, customer files, tax records, databases, marketing plans, or other data are gone forever, the consequences can be catastrophic. While the file system cannot offer any protection against physical destruction of the equipment and media, it can help protect the information. In this section we will look at some of the issues involved in safeguarding the file system.

Floppy disks are generally perfect when they leave the factory, but they can develop bad blocks during use. Hard disks frequently have bad blocks right from the start: it is just too expensive to manufacture them completely free of all defects. As we saw in Chap. 5, bad blocks are generally handled by the controller by replacing bad sectors with spares provided for that purpose. Despite this technique, there are other reliability issues which we will consider below.

Backups

Most people do not think making backups of their files is worth the time and effort—until one fine day their disk abruptly dies, at which time most of them undergo a deathbed conversion. Companies, however, (usually) well understand the value of their data and generally do a backup at least once a day, usually to tape. Modern tapes hold tens or sometimes even hundreds of gigabytes and cost pennies per gigabyte. Nevertheless, Making backups is not quite as trivial as it sounds, so we will examine some of the related issues below.

Spotify USA Inc. Exhibit 1009 Page 128

Backups to tape are generally made to handle one of two potential problems:

1. Recover from disaster.
2. Recover from stupidity.

The first one covers getting the computer running again after a disk crash, fire, flood, or other natural catastrophe. In practice, these things do not happen very often, which is why many people do not bother with backups. These people also tend not to have fire insurance on their houses for the same reason.

The second reason is that users often accidentally remove files that they later need again. This problem occurs so often that when a file is "removed" in Windows, it is not deleted at all, but just moved to a special directory, the **recycle bin**, so it can be fished out and restored easily later. Backups take this principle further and allow files that were removed days, even weeks ago, to be restored from old backup tapes.

Making a backup takes a long time and occupies a large amount of space, so doing it efficiently and conveniently is important. These considerations raise the following issues. First, should the entire file system be backed up or only part of it? At many installations, the executable (binary) programs are kept in a limited part of the file system tree. It is not necessary to back up these files if they can all be reinstalled from the manufacturers' CD-ROMs. Also, most systems have a directory for temporary files. There is usually no reason to back it up either. In UNIX, all the special files (I/O devices) are kept in a directory */dev*. Not only is backing up this directory not necessary, it is downright dangerous because the backup program would hang forever if it tried to read each of these to completion. In short, it is usually desirable to back up only specific directories and everything in them rather than the entire file system.

Second, it is wasteful to back up files that have not changed since the last backup, which leads to the idea of **incremental dumps**. The simplest form of incremental dumping is to make a complete dump (backup) periodically, say weekly or monthly, and to make a daily dump of only those files that have been modified since the last full dump. Even better is to dump only those files that have changed since they were last dumped. While this scheme minimizes dumping time, it makes recovery more complicated because first the most recent full dump has to be restored, followed by all the incremental dumps in reverse order. To ease recovery, more sophisticated incremental dumping schemes are often used.

Third, since immense amounts of data are typically dumped, it may be desirable to compress the data before writing them to tape. However, with many compression algorithms, a single bad spot on the backup tape can foil the decompression algorithm and make an entire file or even an entire tape unreadable. Thus the decision to compress the backup stream must be carefully considered.

Spotify USA Inc. Exhibit 1009 Page 129

Fourth, it is difficult to perform a backup on an active file system. If files and directories are being added, deleted, and modified during the dumping process, the resulting dump may be inconsistent. However, since making a dump may take hours, it may be necessary to take the system offline for much of the night to make the backup, something that is not always acceptable. For this reason, algorithms have been devised for making rapid snapshots of the file system state by copying critical data structures, and then requiring future changes to files and directories to copy the blocks instead of updating them in place (Hutchinson et al., 1999). In this way, the file system is effectively frozen at the moment of the snapshot, so it can be backed up at leisure afterward.

Fifth and last, making backups introduces many nontechnical problems into an organization. The best online security system in the world may be useless if the system administrator keeps all the backup tapes in his office and leaves it open and unguarded whenever he walks down the hall to get output from the printer. All a spy has to do is pop in for a second, put one tiny tape in his pocket, and saunter off jauntily. Goodbye security. Also, making a daily backup has little use if the fire that burns down the computers also burns up all the backup tapes. For this reason, backup tapes should be kept off-site, but that introduces more security risks. For a thorough discussion of these and other practical administration issues, see (Nemeth et al., 2000). Below we will discuss only the technical issues involved in making file system backups.

Two strategies can be used for dumping a disk to tape: a physical dump or a logical dump. A **physical dump** starts at block 0 of the disk, writes all the disk blocks onto the output tape in order, and stops when it has copied the last one. Such a program is so simple that it can probably be made 100% bug free, something that can probably not be said about any other useful program.

Nevertheless, it is worth making several comments about physical dumping. For one thing, there is no value in backing up unused disk blocks. If the dumping program can get access to the free block data structure, it can avoid dumping unused blocks. However, skipping unused blocks requires writing the number of each block in front of the block (or the equivalent), since it is no longer true that block k on the tape was block k on the disk.

A second concern is dumping bad blocks. If all bad blocks are remapped by the disk controller and hidden from the operating system as we described in Sec. 5.4.4, physical dumping works fine. On the other hand, if they are visible to the operating system and maintained in one or more "bad block files" or bitmaps, it is absolutely essential that the physical dumping program get access to this information and avoid dumping them to prevent endless disk read errors during the dumping process.

The main advantages of physical dumping are simplicity and great speed (basically, it can run at the speed of the disk). The main disadvantages are the inability to skip selected directories, make incremental dumps, and restore individual files upon request. For these reasons, most installations make logical dumps.

Spotify USA Inc. Exhibit 1009 Page 130

A **logical dump** starts at one or more specified directories and recursively dumps all files and directories found there that have changed since some given base date (e.g., the last backup for an incremental dump or system installation for a full dump). Thus in a logical dump, the dump tape gets a series of carefully identified directories and files, which makes it easy to restore a specific file or directory upon request.

Since logical dumping is the most common form, let us examine a common algorithm in detail using the example of Fig. 6-24 to guide us. Most UNIX systems use this algorithm. In the figure we see a file tree with directories (squares) and files (circles). The shaded items have been modified since the base date and thus need to be dumped. The unshaded ones do not need to be dumped.

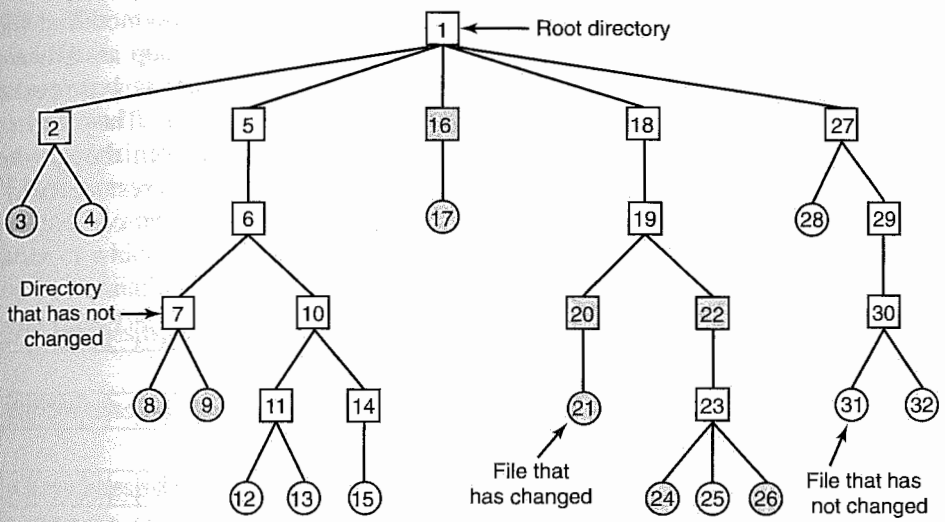


Figure 6-24. A file system to be dumped. The squares are directories and the circles are files. The shaded items have been modified since the last dump. Each directory and file is labeled by its i-node number.

This algorithm also dumps all directories (even unmodified ones) that lie on the path to a modified file or directory for two reasons. First, to make it possible to restore the dumped files and directories to a fresh file system on a different computer. In this way, the dump and restore programs can be used to transport entire file systems between computers.

The second reason for dumping unmodified directories above modified files is to make it possible to incrementally restore a single file (possibly to handle recovery from stupidity). Suppose that a full file system dump is done Sunday evening and an incremental dump is done on Monday evening. On Tuesday the directory */usr/jhs/proj/nr3* is removed, along with all the directories and files under it. On Wednesday morning bright and early the user wants to restore the file */usr/jhs/proj/nr3/plans/summary* However, is not possible to just restore the

file *summary* because there is no place to put it. The directories *nr3* and *plans* must be restored first. To get their owners, modes, times, etc., correct, these directories must be present on the dump tape even though they themselves were not modified since the previous full dump.

The dump algorithm maintains a bitmap indexed by i-node number with several bits per i-node. Bits will be set and cleared in this map as the algorithm proceeds. The algorithm operates in four phases. Phase 1 begins at the starting directory (the root in this example) and examines all the entries in it. For each modified file, its i-node is marked in the bitmap. Each directory is also marked (whether or not it has been modified) and then recursively inspected.

At the end of phase 1, all modified files and all directories have been marked in the bitmap, as shown (by shading) in Fig. 6-25(a). Phase 2 conceptually recursively walks the tree again, unmarking any directories that have no modified files or directories in them or under them. This phase leaves the bitmap as shown in Fig. 6-25(b). Note that directories 10, 11, 14, 27, 29, and 30 are now unmarked because they contain nothing under them that has been modified. They will not be dumped. In contrast, directories 5 and 6 will be dumped even though they themselves have not been modified because they will be needed to restore today's changes to a fresh machine. For efficiency, phases 1 and 2 can be combined in one tree walk.

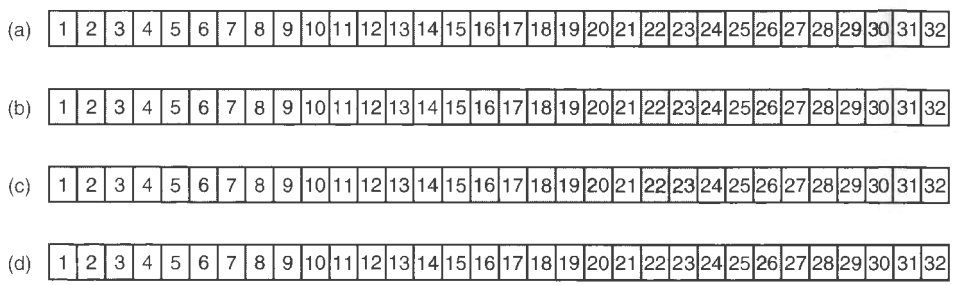


Figure 6-25. Bit maps used by the logical dumping algorithm.

At this point it is known which directories and files must be dumped. These are the ones marked in Fig. 6-25(b). Phase 3 consists of scanning the i-nodes in numerical order and dumping all the directories that are marked for dumping. These are shown in Fig. 6-25(c). Each directory is prefixed by the directory's attributes (owner, times, etc.) so they can be restored. Finally, in phase 4, the files marked in Fig. 6-25(d) are also dumped, again prefixed by their attributes. This completes the dump.

Restoring a file system from the dump tapes is straightforward. To start with, an empty file system is created on the disk. Then the most recent full dump is restored. Since the directories appear first on the tape, they are all restored first,

Spotify USA Inc. Exhibit 1009 Page 132

giving a skeleton of the file system. Then the files themselves are restored. This process is then repeated with the first incremental dump made after the full dump, then the next one, and so on.

Although logical dumping is straightforward, there are a few tricky issues. For one, since the free block list is not a file, it is not dumped and hence it must be reconstructed from scratch after all the dumps have been restored. Doing so is always possible since the set of free blocks is just the complement of the set of blocks contained in all the files combined.

Another issue is links. If a file is linked to two or more directories, it is important that the file is restored only one time and that all the directories that are supposed to point to it do so.

Still another issue is the fact that UNIX files may contain holes. It is legal to open a file, write a few bytes, then seek to a distant file offset and write a few more bytes. The blocks in between are not part of the file and should not be dumped and not be restored. Core files often have a large hole between the data segment and the stack. If not handled properly, each restored core file will fill this area with zeros and thus be the same size as the virtual address space (e.g., 2^{32} bytes, or worse yet, 2^{64} bytes).

Finally, special files, named pipes, and the like should never be dumped, no matter in which directory they may occur (they need not be confined to */dev*). For more information about file system backups, see (Chervenak et al., 1998; and Zwicky, 1991).

File System Consistency

Another area where reliability is an issue is file system consistency. Many file systems read blocks, modify them, and write them out later. If the system crashes before all the modified blocks have been written out, the file system can be left in an inconsistent state. This problem is especially critical if some of the blocks that have not been written out are i-node blocks, directory blocks, or blocks containing the free list.

To deal with the problem of inconsistent file systems, most computers have a utility program that checks file system consistency. For example, UNIX has *fsck* and Windows has *scandisk*. This utility can be run whenever the system is booted, especially after a crash. The description below tells how *fsck* works. *Scandisk* is somewhat different because it works on a different file system, but the general principle of using the file system's inherent redundancy to repair it is still valid. All file system checkers verify each file system (disk partition) independently of the other ones.

Two kinds of consistency checks can be made: blocks and files. To check for block consistency, the program builds two tables, each one containing a counter for each block, initially set to 0. The counters in the first table keep track of how

Spotify USA Inc. Exhibit 1009 Page 133

many times each block is present in a file; the counters in the second table record how often each block is present in the free list (or the bitmap of free blocks).

The program then reads all the i-nodes. Starting from an i-node, it is possible to build a list of all the block numbers used in the corresponding file. As each block number is read, its counter in the first table is incremented. The program then examines the free list or bitmap, to find all the blocks that are not in use. Each occurrence of a block in the free list results in its counter in the second table being incremented.

If the file system is consistent, each block will have a 1 either in the first table or in the second table, as illustrated in Fig. 6-26(a). However, as a result of a crash, the tables might look like Fig. 6-26(b), in which block 2 does not occur in either table. It will be reported as being a **missing block**. While missing blocks do no real harm, they do waste space and thus reduce the capacity of the disk. The solution to missing blocks is straightforward: the file system checker just adds them to the free list.

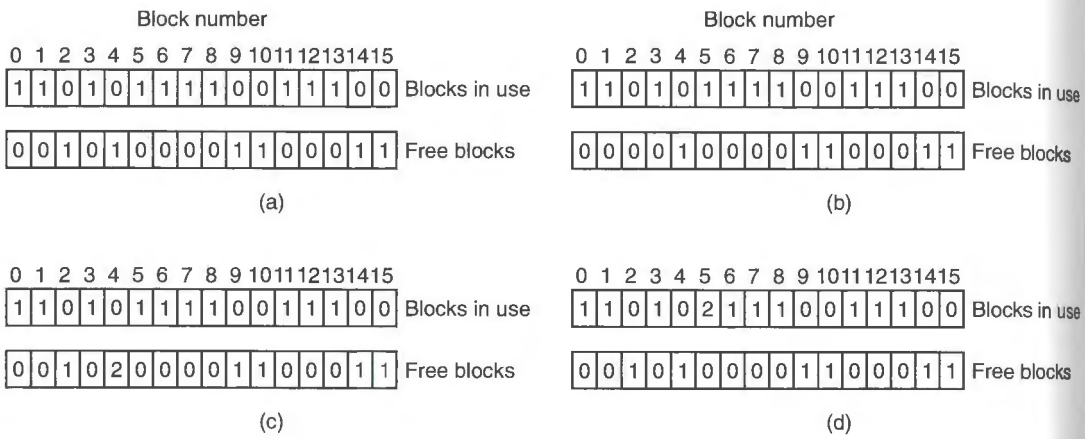


Figure 6-26. File system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

Another situation that might occur is that of Fig. 6-26(c). Here we see a block, number 4, that occurs twice in the free list. (Duplicates can occur only if the free list is really a list; with a bitmap it is impossible.) The solution here is also simple: rebuild the free list.

The worst thing that can happen is that the same data block is present in two or more files, as shown in Fig. 6-26(d) with block 5. If either of these files is removed, block 5 will be put on the free list, leading to a situation in which the same block is both in use and free at the same time. If both files are removed, the block will be put onto the free list twice.

The appropriate action for the file system checker to take is to allocate a free block, copy the contents of block 5 into it, and insert the copy into one of the files.

Spotify USA Inc. Exhibit 1009 Page 134

In this way, the information content of the files is unchanged (although almost assuredly one is garbled), but the file system structure is at least made consistent. The error should be reported, to allow the user to inspect the damage.

In addition to checking to see that each block is properly accounted for, the file system checker also checks the directory system. It, too, uses a table of counters, but these are per file, rather than per block. It starts at the root directory and recursively descends the tree, inspecting each directory in the file system. For every file in every directory, it increments a counter for that file's usage count. Remember that due to hard links, a file may appear in two or more directories. Symbolic links do not count and do not cause the counter for the target file to be incremented.

When it is all done, it has a list, indexed by i-node number, telling how many directories contain each file. It then compares these numbers with the link counts stored in the i-nodes themselves. These counts start at 1 when a file is created and are incremented each time a (hard) link is made to the file. In a consistent file system, both counts will agree. However, two kinds of errors can occur: the link count in the i-node can be too high or it can be too low.

If the link count is higher than the number of directory entries, then even if all the files are removed from the directories, the count will still be nonzero and the i-node will not be removed. This error is not serious, but it wastes space on the disk with files that are not in any directory. It should be fixed by setting the link count in the i-node to the correct value.

The other error is potentially catastrophic. If two directory entries are linked to a file, but the i-node says that there is only one, when either directory entry is removed, the i-node count will go to zero. When an i-node count goes to zero, the file system marks it as unused and releases all of its blocks. This action will result in one of the directories now pointing to an unused i-node, whose blocks may soon be assigned to other files. Again, the solution is just to force the link count in the i-node to the actual number of directory entries.

These two operations, checking blocks and checking directories, are often integrated for efficiency reasons (i.e., only one pass over the i-nodes is required). Other checks are also possible. For example, directories have a definite format, with i-node numbers and ASCII names. If an i-node number is larger than the number of i-nodes on the disk, the directory has been damaged.

Furthermore, each i-node has a mode, some of which are legal but strange, such as 0007, which allows the owner and his group no access at all, but allows outsiders to read, write, and execute the file. It might be useful to at least report files that give outsiders more rights than the owner. Directories with more than, say, 1000 entries are also suspicious. Files located in user directories, but which are owned by the superuser and have the SETUID bit on, are potential security problems because such files acquire the powers of the superuser when executed by any user. With a little effort, one can put together a fairly long list of technically legal but still peculiar situations that might be worth reporting.

Spotify USA Inc. Exhibit 1009 Page 135

The previous paragraphs have discussed the problem of protecting the user against crashes. Some file systems also worry about protecting the user against himself. If the user intends to type

```
rm *.o
```

to remove all the files ending with *.o* (compiler generated object files), but accidentally types

```
rm * .o
```

(note the space after the asterisk), *rm* will remove all the files in the current directory and then complain that it cannot find *.o*. In MS-DOS and some other systems, when a file is removed, all that happens is that a bit is set in the directory or i-node marking the file as removed. No disk blocks are returned to the free list until they are actually needed. Thus, if the user discovers the error immediately, it is possible to run a special utility program that "unremoves" (i.e., restores) the removed files. In Windows, files that are removed are placed in the recycle bin, from which they can later be retrieved if need be. Of course, no storage is reclaimed until they are actually deleted from this directory.

6.3.7 File System Performance

Access to disk is much slower than access to memory. Reading a memory word might take 10 nsec. Reading from a hard disk might proceed at 10 MB/sec, which is forty times slower per 32-bit word, but to this must be added 5–10 msec to seek to the track and then wait for the desired sector to arrive under the read head. If only a single word is needed, the memory access is on the order of a million times as fast as disk access. As a result of this difference in access time, many file systems have been designed with various optimizations to improve performance. In this section we will cover three of them.

Caching

The most common technique used to reduce disk accesses is the **block cache** or **buffer cache**. (Cache is pronounced "cash" and is derived from the French *cacher*, meaning to hide.) In this context, a cache is a collection of blocks that logically belong on the disk but are being kept in memory for performance reasons.

Various algorithms can be used to manage the cache, but a common one is to check all read requests to see if the needed block is in the cache. If it is, the read request can be satisfied without a disk access. If the block is not in the cache, it is first read into the cache, and then copied to wherever it is needed. Subsequent requests for the same block can be satisfied from the cache.

Operation of the cache is illustrated in Fig. 6-27. Since there are many (often thousands of) blocks in the cache, some way is needed to determine quickly if a

given block is present. The usual way is to hash the device and disk address and look up the result in a hash table. All the blocks with the same hash value are chained together on a linked list so the collision chain can be followed.

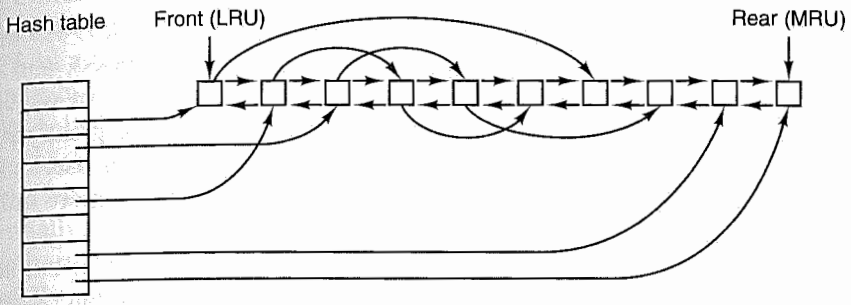


Figure 6-27. The buffer cache data structures.

When a block has to be loaded into a full cache, some block has to be removed (and rewritten to the disk if it has been modified since being brought in). This situation is very much like paging, and all the usual page replacement algorithms described in Chap. 3, such as FIFO, second chance, and LRU, are applicable. One pleasant difference between paging and caching is that cache references are relatively infrequent, so that it is feasible to keep all the blocks in exact LRU order with linked lists.

In Fig. 6-27, we see that in addition to the collision chains starting at the hash table, there is also a bidirectional list running through all the blocks in the order of usage, with the least recently used block on the front of this list and the most recently used block at the end of this list. When a block is referenced, it can be removed from its position on the bidirectional list and put at the end. In this way, exact LRU order can be maintained.

Unfortunately, there is a catch. Now that we have a situation in which exact LRU is possible, it turns out that LRU is undesirable. The problem has to do with the crashes and file system consistency discussed in the previous section. If a critical block, such as an i-node block, is read into the cache and modified, but not rewritten to the disk, a crash will leave the file system in an inconsistent state. If the i-node block is put at the end of the LRU chain, it may be quite a while before it reaches the front and is rewritten to the disk.

Furthermore, some blocks, such as i-node blocks, are rarely referenced two times within a short interval. These considerations lead to a modified LRU scheme, taking two factors into account:

1. Is the block likely to be needed again soon?
2. Is the block essential to the consistency of the file system?

For both questions, blocks can be divided into categories such as i-node blocks, indirect blocks, directory blocks, full data blocks, and partially full data blocks.

Spotify USA Inc. Exhibit 1009 Page 137

Blocks that will probably not be needed again soon go on the front, rather than the rear of the LRU list, so their buffers will be reused quickly. Blocks that might be needed again soon, such as a partly full block that is being written, go on the end of the list, so they will stay around for a long time.

The second question is independent of the first one. If the block is essential to the file system consistency (basically, everything except data blocks), and it has been modified, it should be written to disk immediately, regardless of which end of the LRU list it is put on. By writing critical blocks quickly, we greatly reduce the probability that a crash will wreck the file system. While a user may be unhappy if one of his files is ruined in a crash, he is likely to be far more unhappy if the whole file system is lost.

Even with this measure to keep the file system integrity intact, it is undesirable to keep data blocks in the cache too long before writing them out. Consider the plight of someone who is using a personal computer to write a book. Even if our writer periodically tells the editor to write the file being edited to the disk, there is a good chance that everything will still be in the cache and nothing on the disk. If the system crashes, the file system structure will not be corrupted, but a whole day's work will be lost.

This situation need not happen very often before we have a fairly unhappy user. Systems take two approaches to dealing with it. The UNIX way is to have a system call, sync, which forces all the modified blocks out onto the disk immediately. When the system is started up, a program, usually called *update*, is started up in the background to sit in an endless loop issuing sync calls, sleeping for 30 sec between calls. As a result, no more than 30 seconds of work is lost due to a crash.

The MS-DOS way is to write every modified block to disk as soon as it has been written. Caches in which all modified blocks are written back to the disk immediately are called **write-through caches**. They require more disk I/O than nonwrite-through caches. The difference between these two approaches can be seen when a program writes a 1-KB block full, one character at a time. UNIX will collect all the characters in the cache and write the block out once every 30 seconds, or whenever the block is removed from the cache. MS-DOS will make a disk access for every character written. Of course, most programs do internal buffering, so they normally write not a character, but a line or a larger unit on each write system call.

A consequence of this difference in caching strategy is that just removing a (floppy) disk from a UNIX system without doing a sync will almost always result in lost data, and frequently in a corrupted file system as well. With MS-DOS, no problem arises. These differing strategies were chosen because UNIX was developed in an environment in which all disks were hard disks and not removable, whereas MS-DOS started out in the floppy disk world. As hard disks became the norm, the UNIX approach, with its better efficiency, became the norm, and is also used now on Windows for hard disks.

Spotify USA Inc. Exhibit 1009 Page 138

Block Read Ahead

A second technique for improving perceived file system performance is to try to get blocks into the cache before they are needed to increase the hit rate. In particular, many files are read sequentially. When the file system is asked to produce block k in a file, it does that, but when it is finished, it makes a sneaky check in the cache to see if block $k + 1$ is already there. If it is not, it schedules a read for block $k + 1$ in the hope that when it is needed, it will have already arrived in the cache. At the very least, it will be on the way.

Of course, this read ahead strategy only works for files that are being read sequentially. If a file is being randomly accessed, read ahead does not help. In fact, it hurts by tying up disk bandwidth reading in useless blocks and removing potentially useful blocks from the cache (and possibly tying up more disk bandwidth writing them back to disk if they are dirty). To see whether read ahead is worth doing, the file system can keep track of the access patterns to each open file. For example, a bit associated with each file can keep track of whether the file is in "sequential access mode" or "random access mode." Initially, the file is given the benefit of the doubt and put in sequential access mode. However, whenever a seek is done, the bit is cleared. If sequential reads start happening again, the bit is set once again. In this way, the file system can make a reasonable guess about whether it should read ahead or not. If it gets it wrong once it a while, it is not a disaster, just a little bit of wasted disk bandwidth.

Reducing Disk Arm Motion

Caching and read ahead are not the only ways to increase file system performance. Another important technique is to reduce the amount of disk arm motion by putting blocks that are likely to be accessed in sequence close to each other, preferably in the same cylinder. When an output file is written, the file system has to allocate the blocks one at a time, as they are needed. If the free blocks are recorded in a bitmap, and the whole bitmap is in main memory, it is easy enough to choose a free block as close as possible to the previous block. With a free list, part of which is on disk, it is much harder to allocate blocks close together.

However, even with a free list, some block clustering can be done. The trick is to keep track of disk storage not in blocks, but in groups of consecutive blocks. If sectors consist of 512 bytes, the system could use 1-KB blocks (2 sectors) but allocate disk storage in units of 2 blocks (4 sectors). This is not the same as having a 2-KB disk blocks, since the cache would still use 1-KB blocks and disk transfers would still be 1 KB but reading a file sequentially on an otherwise idle system would reduce the number of seeks by a factor of two, considerably improving performance. A variation on the same theme is to take account of rotational positioning. When allocating blocks, the system attempts to place consecutive blocks in a file in the same cylinder.

Spotify USA Inc. Exhibit 1009 Page 139

Another performance bottleneck in systems that use i-nodes or anything equivalent to i-nodes is that reading even a short file requires two disk accesses: one for the i-node and one for the block. The usual i-node placement is shown in Fig. 6-28(a). Here all the i-nodes are near the beginning of the disk, so the average distance between an i-node and its blocks will be about half the number of cylinders, requiring long seeks.

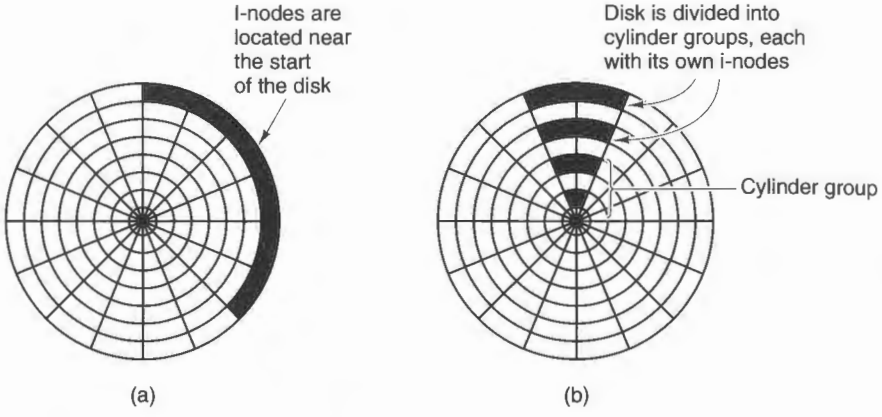


Figure 6-28. (a) I-nodes placed at the start of the disk. (b) Disk divided into cylinder groups, each with its own blocks and i-nodes.

One easy performance improvement is to put the i-nodes in the middle of the disk, rather than at the start, thus reducing the average seek between the i-node and the first block by a factor of two. Another idea, shown in Fig. 6-28(b), is to divide the disk into cylinder groups, each with its own i-nodes, blocks, and free list (McKusick et al., 1984). When creating a new file, any i-node can be chosen, but an attempt is made to find a block in the same cylinder group as the i-node. If none is available, then a block in a nearby cylinder group is used.

6.3.8 Log-Structured File Systems

Changes in technology are putting pressure on current file systems. In particular, CPUs keep getting faster, disks are becoming much bigger and cheaper (but not much faster), and memories are growing exponentially in size. The one parameter that is not improving by leaps and bounds is disk seek time. The combination of these factors means that a performance bottleneck is arising in many file systems. Research done at Berkeley attempted to alleviate this problem by designing a completely new kind of file system, LFS (the **Log-structured File System**). In this section we will briefly describe how LFS works. For a more complete treatment, see (Rosenblum and Ousterhout, 1991).

The idea that drove the LFS design is that as CPUs get faster and RAM memories get larger, disk caches are also increasing rapidly. Consequently, it is now

Spotify USA Inc. Exhibit 1009 Page 140

possible to satisfy a very substantial fraction of all read requests directly from the file system cache, with no disk access needed. It follows from this observation, that in the future, most disk accesses will be writes, so the read-ahead mechanism used in some file systems to fetch blocks before they are needed no longer gains much performance.

To make matters worse, in most file systems, writes are done in very small chunks. Small writes are highly inefficient, since a 50-μsec disk write is often preceded by a 10-msec seek and a 4-msec rotational delay. With these parameters, disk efficiency drops to a fraction of 1 percent.

To see where all the small writes come from, consider creating a new file on a UNIX system. To write this file, the i-node for the directory, the directory block, the i-node for the file, and the file itself must all be written. While these writes can be delayed, doing so exposes the file system to serious consistency problems if a crash occurs before the writes are done. For this reason, the i-node writes are generally done immediately.

From this reasoning, the LFS designers decided to re-implement the UNIX file system in such a way as to achieve the full bandwidth of the disk, even in the face of a workload consisting in large part of small random writes. The basic idea is to structure the entire disk as a log. Periodically, and when there is a special need for it, all the pending writes being buffered in memory are collected into a single segment and written to the disk as a single contiguous segment at the end of the log. A single segment may thus contain i-nodes, directory blocks, and data blocks, all mixed together. At the start of each segment is a segment summary, telling what can be found in the segment. If the average segment can be made to be about 1 MB, almost the full bandwidth of the disk can be utilized.

In this design, i-nodes still exist and have the same structure as in UNIX, but they are now scattered all over the log, instead of being at a fixed position on the disk. Nevertheless, when an i-node is located, locating the blocks is done in the usual way. Of course, finding an i-node is now much harder, since its address cannot simply be calculated from its i-number, as in UNIX. To make it possible to find i-nodes, an i-node map, indexed by i-number, is maintained. Entry *i* in this map points to i-node *i* on the disk. The map is kept on disk, but it is also cached, so the most heavily used parts will be in memory most of the time.

To summarize what we have said so far, all writes are initially buffered in memory, and periodically all the buffered writes are written to the disk in a single segment, at the end of the log. Opening a file now consists of using the map to locate the i-node for the file. Once the i-node has been located, the addresses of the blocks can be found from it. All of the blocks will themselves be in segments, somewhere in the log.

If disks were infinitely large, the above description would be the entire story. However, real disks are finite, so eventually the log will occupy the entire disk, at which time no new segments can be written to the log. Fortunately, many existing segments may have blocks that are no longer needed, for example, if a file is

Spotify USA Inc. Exhibit 1009 Page 141

overwritten, its i-node will now point to the new blocks, but the old ones will still be occupying space in previously written segments.

To deal with this problem, LFS has a **cleaner** thread that spends its time scanning the log circularly to compact it. It starts out by reading the summary of the first segment in the log to see which i-nodes and files are there. It then checks the current i-node map to see if the i-nodes are still current and file blocks are still in use. If not, that information is discarded. The i-nodes and blocks that are still in use go into memory to be written out in the next segment. The original segment is then marked as free, so the log can use it for new data. In this manner, the cleaner moves along the log, removing old segments from the back and putting any live data into memory for rewriting in the next segment. Consequently, the disk is a big circular buffer, with the writer thread adding new segments to the front and the cleaner thread removing old ones from the back.

The bookkeeping here is nontrivial, since when a file block is written back to a new segment, the i-node of the file (somewhere in the log) must be located, updated, and put into memory to be written out in the next segment. The i-node map must then be updated to point to the new copy. Nevertheless, it is possible to do the administration, and the performance results show that all this complexity is worthwhile. Measurements given in the papers cited above show that LFS outperforms UNIX by an order of magnitude on small writes, while having a performance that is as good as or better than UNIX for reads and large writes.

6.4 EXAMPLE FILE SYSTEMS

In the following sections we will discuss several example file systems, ranging from quite simple to highly sophisticated. Since modern UNIX file systems and Windows 2000's native file system are covered in the chapter on UNIX (Chap. 10) and the chapter on Windows 2000 (Chap. 11) we will not cover those systems here. We will, however, examine their predecessors below.

6.4.1 CD-ROM File Systems

As our first example of a file system, let us consider the file systems used on CD-ROMs. These systems are particularly simple because they were designed for write-once media. Among other things, for example, they have no provision for keeping track of free blocks because on a CD-ROM files cannot be freed or added after the disk has been manufactured. Below we will take a look at the main CD-ROM file system type and two extensions to it.

The ISO 9660 File System

The most common standard for CD-ROM file systems was adopted as an International Standard in 1988 under the name **ISO 9660**. Virtually every CD-ROM currently on the market is compatible with this standard, sometimes with

Spotify USA Inc. Exhibit 1009 Page 142

the extensions to be discussed below. One of the goals of this standard was to make every CD-ROM readable on every computer, independent of the byte ordering used and independent of the operating system used. As a consequence, some limitations were placed on the file system to make it possible for the weakest operating systems then in use (such as MS-DOS) to read it.

CD-ROMs do not have concentric cylinders the way magnetic disks do. Instead there is a single continuous spiral containing the bits in a linear sequence (although seeks across the spiral are possible). The bits along the spiral are divided into logical blocks (also called logical sectors) of 2352 bytes. Some of these are for preambles, error correction, and other overhead. The payload portion of each logical block is 2048 bytes. When used for music, CDs have leadins, leadouts, and intertrack gaps, but these are not used for data CD-ROMs. Often the position of a block along the spiral is quoted in minutes and seconds. It can be converted to a linear block number using the conversion factor of 1 sec = 75 blocks.

ISO 9660 supports CD-ROM sets with as many as $2^{16} - 1$ CDs in the set. The individual CD-ROMs may also be partitioned into logical volumes (partitions). However, below we will concentrate on ISO 9660 for a single unpartitioned CD-ROM.

Every CD-ROM begins with 16 blocks whose function is not defined by the ISO 9660 standard. A CD-ROM manufacturer could use this area for providing a bootstrap program to allow the computer to be booted from the CD-ROM, or for some other purpose. Next comes one block containing the **primary volume descriptor**, which contains some general information about the CD-ROM. Among this information are the system identifier (32 bytes), volume identifier (32 bytes), publisher identifier (128 bytes), and data preparer identifier (128 bytes). The manufacturer can fill in these fields in any desired way, except that only upper case letters, digits, and a very small number of punctuation marks may be used to ensure cross-platform compatibility.

The primary volume descriptor also contains the names of three files, which may contain the abstract, copyright notice, and bibliographic information, respectively. In addition, certain key numbers are also present, including the logical block size (normally 2048, but 4096, 8192, and larger powers of two are allowed in certain cases), the number of blocks on the CD-ROM, and the creation and expiration dates of the CD-ROM. Finally, the primary volume descriptor also contains a directory entry for the root directory, telling where to find it on the CD-ROM (i.e., which block it starts at). From this directory, the rest of the file system can be located.

In addition to the primary volume descriptor, a CD-ROM may contain a supplementary volume descriptor. It contains similar information to the primary, but that will not concern us here.

The root directory, and all other directories for that matter, consists of a variable number of entries, the last of which contains a bit marking it as the final one.

Spotify USA Inc. Exhibit 1009 Page 143

The directory entries themselves are also variable length. Each directory entry consists of 10 to 12 fields, some of which are in ASCII and others of which are numerical fields in binary. The binary fields are encoded twice, once in little-endian format (used on Pentiums, for example) and once in big-endian format (used on SPARCs, for example). Thus a 16-bit number uses 4 bytes and a 32-bit number uses 8 bytes. The use of this redundant coding was necessary to avoid hurting anyone's feelings when the standard was developed. If the standard had dictated little endian, then people from companies with big-endian products would have felt like second-class citizens and would not have accepted the standard. The emotional content of a CD-ROM can thus be quantified and measured exactly in kilobytes/hour of wasted space.

The format of an ISO 9660 directory entry is illustrated in Fig. 6-29. Since directory entries have variable lengths, the first field is a byte telling how long the entry is. This byte is defined to have the high-order bit on the left to avoid any ambiguity.

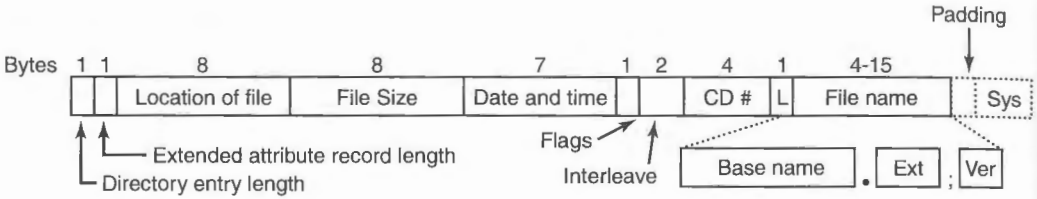


Figure 6-29. The ISO 9660 directory enty.

Directory entries may optionally have an extended attributes. If this feature is used for a directory entry, the second byte tells how long the extended attributes are.

Next comes the starting block of the file itself. Files are stored as contiguous runs of blocks, so a file's location is completely specified by the starting block and the size, which is contained in the next field.

The date and time that the CD-ROM was recorded is stored in the next field, with separate bytes for the year, month, day, hour, minute, second, and time zone. Years begin to count at 1900, which means that CD-ROMs will suffer from a Y2156 problem because the year following 2155 will be 1900. This problem could have been delayed by defining the origin of time to be 1988 (the year the standard was adopted). Had that been done, the problem would have been postponed until 2244. Every 88 extra years helps.

The *Flags* field contains a few miscellaneous bits, including one to hide the entry in listings (a feature copied from MS-DOS), one to distinguish an entry that is a file from an entry that is a directory, one to enable the use of the extended attributes, and one to mark the last entry in a directory. A few other bits are also

Spotify USA Inc. Exhibit 1009 Page 144

present in this field but they will not concern us here. The next field deals with interleaving pieces of files in a way that is not used in the simplest version of ISO 9660, so we will not consider it further.

The next field tells which CD-ROM the file is located on. It is permitted that a directory entry on one CD-ROM refers to a file located on another CD-ROM in the set. In this way it is possible to build a master directory on the first CD-ROM that lists all the files on all the CD-ROMs in the complete set.

The field marked *L* in Fig. 6-29 gives the size of the file name in bytes. It is followed by the file name itself. A file name consists of a base name, a dot, an extension, a semicolon, and a binary version number (1 or 2 bytes). The base name and extension may use upper case letters, the digits 0–9, and the underscore character. All other characters are forbidden to make sure that every computer can handle every file name. The base name can be up to eight characters; the extension can be up to three characters. These choices were dictated by the need to be MS-DOS compatible. A file name may be present in a directory multiple times, as long as each one has a different version number.

The last two fields are not always present. The *Padding* field is used to force every directory entry to be an even number of bytes, to align the numeric fields of subsequent entries on 2-byte boundaries. If padding is needed, a 0 byte is used. Finally, we have the *System use* field. Its function and size are undefined, except that it must be an even number of bytes. Different systems use it in different ways. The Macintosh keeps Finder flags here, for example.

Entries within a directory are listed in alphabetical order except for the first two entries. The first entry is for the directory itself. The second one is for its parent. In this respect, these entries are similar to the UNIX . and .. directory entries. The files themselves need not be in directory order.

There is no explicit limit to the number of entries in a directory. However, there is a limit to the depth of nesting. The maximum depth of directory nesting is eight.

ISO 9660 defines what are called three levels. Level 1 is the most restrictive and specifies that file names are limited to 8 + 3 characters as we have described, and also requires all files to be contiguous as we have described. Furthermore, it specifies that directory names be limited to eight characters with no extensions. Use of this level maximizes the chances that a CD-ROM can be read on every computer.

Level 2 relaxes the length restriction. It allows files and directories to have names of up to 31 characters, but still from the same set of characters.

Level 3 uses the same name limits as level 2, but partially relaxes the assumption that files have to be contiguous. With this level, a file may consist of several sections, each of which is a contiguous run of blocks. The same run may occur multiple times in a file and may also occur in two or more files. If large chunks of data are repeated in several files, level 3 provides some space optimization by not requiring the data to be present multiple times.

Spotify USA Inc. Exhibit 1009 Page 145

Rock Ridge Extensions

As we have seen, ISO 9660 is highly restrictive in several ways. Shortly after it came out, people in the UNIX community began working on an extension to make it possible to represent UNIX file systems on a CD-ROM. These extensions were named Rock Ridge, after a town in the Gene Wilder movie *Blazing Saddles*, probably because one of the committee members liked the film.

The extensions use the *System use* field in order to make Rock Ridge CD-ROMs readable on any computer. All the other fields retain their normal ISO 9660 meaning. Any system not aware of the Rock Ridge extensions just ignores them and sees a normal CD-ROM.

The extensions are divided up into the following fields:

1. PX - POSIX attributes.
2. PN - Major and minor device numbers.
3. SL - Symbolic link.
4. NM - Alternative name.
5. CL - Child location.
6. PL - Parent location.
7. RE - Relocation.
8. TF - Time stamps.

The *PX* field contains the standard UNIX *rwxrwxrwx* permission bits for the owner, group, and others. It also contains the other bits contained in the mode word, such as the SETUID and SETGID bits, and so on.

To allow raw devices to be represented on a CD-ROM, the *PN* field is present. It contains the major and minor device numbers associated with the file. In this way, the contents of the */dev* directory can be written to a CD-ROM and later reconstructed correctly on the target system.

The *SL* field is for symbolic links. It allows a file on one file system to refer to a file on a different file system.

Probably the most important field is *NM*. It allows a second name to be associated with the file. This name is not subject to the character set or length restrictions of ISO 9660, making it possible to express arbitrary UNIX file names on a CD-ROM.

The next three fields are used together to get around the ISO 9660 limit of directories that may only be nested eight deep. Using them it is possible to specify that a directory is to be relocated, and to tell where it goes in the hierarchy. It is effectively a way to work around the artificial depth limit.

Spotify USA Inc. Exhibit 1009 Page 146

Finally, the *TF* field contains the three timestamps included in each UNIX i-node, namely the time the file was created, the time it was last modified, and the time it was last accessed. Together, these extensions make it possible to copy a UNIX file system to a CD-ROM and then restore it correctly to a different system.

Joliet Extensions

The UNIX community was not the only group that wanted a way to extend ISO 9660. Microsoft also found it too restrictive (although it was Microsoft's own MS-DOS that caused most of the restrictions in the first place). Therefore Microsoft invented some extensions that were called **Joliet**. They were designed to allow Windows file systems to be copied to CD-ROM and then restored, in precisely the same way that Rock Ridge was designed for UNIX. Virtually all programs that run under Windows and use CD-ROMs support Joliet, including programs that burn CD-recordables. Usually, these programs offer a choice between the various ISO 9660 levels and Joliet.

The major extensions provided by Joliet are

1. Long file names.
2. Unicode character set.
3. Directory nesting deeper than eight levels.
4. Directory names with extensions

The first extension allows file names up to 64 characters. The second extension enables the use of the Unicode character set for file names. This extension is important for software intended for use in countries that do not use the Latin alphabet, such as Japan, Israel, and Greece. Since Unicode characters are two bytes, the maximum file name in Joliet occupies 128 bytes.

Like Rock Ridge, the limitation on directory nesting is removed by Joliet. Directories can be nested as deeply as needed. Finally, directory names can have extensions. It is not clear why this extension was included, since Windows directories virtually never use extensions, but maybe some day they will.

6.4.2 The CP/M File System

The first personal computers (then called microcomputers) came out in the early 1980s. A popular early type used the 8-bit Intel 8080 CPU and had 4 KB of RAM and a single 8-inch floppy disk with a capacity of 180 KB. Later versions used the slightly fancier (but still 8-bit) Zilog Z80 CPU, had up to 64 KB of RAM, and had a whopping 720-KB floppy disk as the mass storage device. Despite the slow speed and small amount of RAM, nearly all of these machines ran a surprisingly powerful disk-based operating system, called **CP/M** (**Control**

Spotify USA Inc. Exhibit 1009 Page 147

Program for Microcomputers) (Golden and Pechura, 1986). This system dominated its era as much as MS-DOS and later Windows dominated the IBM PC world. Two decades later, it has vanished without a trace (except for a small group of diehard fans), which gives reason to think that systems that now dominate the world may be essentially unknown when current babies become college students (Windows what?).

It is worth taking a look at CP/M for several reasons. First, it was a historically very important system and was the direct ancestor of MS-DOS. Second, current and future operating system designers who think that a computer needs 32 MB just to boot the operating system could probably learn a lot about simplicity from a system that ran quite well in 16 KB of RAM. Third, in the coming decades, embedded systems are going to be extremely widespread. Due to cost, space, weight, and power constraints, the operating systems used, for example, in watches, cameras, radios, and cellular telephones, are of necessity going to be lean and mean, not unlike CP/M. Of course, these systems do not have 8-inch floppy disks, but they may well have electronic disks using flash memory, and building a CP/M-like file system on such a device is straightforward.

The layout of CP/M in memory is shown in Fig. 6-30. At the top of main memory (in RAM) is the BIOS, which contains a basic library of 17 I/O calls used by CP/M (in this section we will describe CP/M 2.2, which was the standard version when CP/M was at the height of its popularity). These calls read and write the keyboard, screen, and floppy disk.

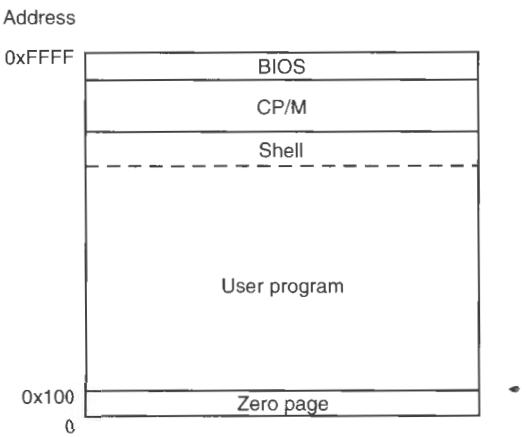


Figure 6-30. Memory layout of CP/M.

Just below the BIOS is the operating system proper. The size of the operating system in CP/M 2.2 is 3584 bytes. Amazing but true: a complete operating system in under 4 KB. Below the operating system comes the shell (command line processor), which chews up another 2 KB. The rest of memory is for user programs, except for the bottom 256 bytes, which are reserved for the hardware interrupt

Spotify USA Inc. Exhibit 1009 Page 148

vectors, a few variables, and a buffer for the current command line so user programs can get at it.

The reason for splitting the BIOS from CP/M itself (even though both are in RAM) is portability. CP/M interacts with the hardware only by making BIOS calls. To port CP/M to a new machine, all that is necessary is to port the BIOS there. Once that has been done, CP/M itself can be installed without modification.

A CP/M system has only one directory, which contains fixed-size (32-byte) entries. The directory size, although fixed for a given implementation, may be different in other implementations of CP/M All files in the system are listed in this directory. After CP/M boots, it reads in the directory and computes a bitmap containing the free disk blocks by seeing which blocks are not in any file. This bitmap, which is only 23 bytes for a 180-KB disk, is kept in memory during execution. At system shutdown time it is discarded, that is, not written back to the disk. This approach eliminates the need for a disk consistency checker (like *fsck*) and saves 1 block on the disk (percentually equivalent to saving 90 MB on a modern 16-GB disk).

When the user types a command, the shell first copies the command to a buffer in the bottom 256 bytes of memory. Then it looks up the program to be executed and reads it into memory at address 256 (above the interrupt vectors), and jumps to it. The program then begins running. It discovers its arguments by looking in the command line buffer. The program is allowed to overwrite the shell if it needs the memory. When the program finishes, it makes a system call to CP/M telling it to reload the shell (if it was overwritten) and execute it. In a nutshell, that is pretty much the whole CP/M story.

In addition to loading programs, CP/M provides 38 system calls, mostly file services, for user programs. The most important of these are reading and writing files. Before a file can be read, it must be opened. When CP/M gets an open system call, it has to read in and search the one and only directory. The directory is not kept in memory all the time to save precious RAM. When CP/M finds the entry, it immediately has the disk block numbers, since they are stored right in the directory entry, as are all the attributes. The format of a directory entry is given in Fig. 6-31.

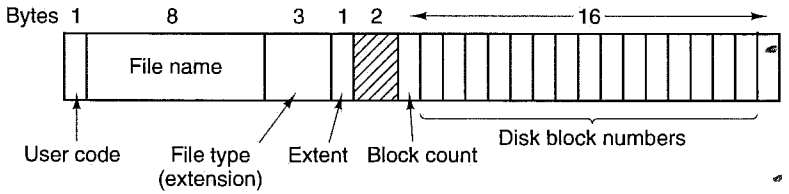


Figure 6-31. The CP/M directory entry format.

The fields in Fig. 6-31 have the following meanings. The *User code* field keeps track of which user owns the file. Although only one person can be logged

into a CP/M at any given moment, the system supports multiple users who take turns using the system. While searching for a file name, only those entries belonging to the currently logged-in user are checked. In effect, each user has a virtual directory without the overhead of managing multiple directories.

The next two fields give the name and extension of the file. The base name is up to eight characters; an optional extension of up to three characters may be present. Only upper case letters, digits, and a small number of special characters are allowed in file names. This 8 + 3 naming using upper case only was later taken over by MS-DOS.

The *Block count* field tells how many bytes this file entry contains, measured in units of 128 bytes (because I/O is actually done in 128-byte physical sectors). The last 1-KB block may not be full, so the system has no way to determine the exact size of a file. It is up to the user to put in some END-OF-FILE marker if desired. The final 16 fields contain the disk block numbers themselves. Each block is 1 KB, so that maximum file size is 16 KB. Note that physical I/O is done in units of 128-byte sectors and sizes are kept track of in sectors, but file blocks are allocated in units of 1 KB (8 sectors at a time) to avoid making the directory entry too large.

However, the CP/M designer realized that some files, even on a 180-KB floppy disk, might exceed 16 KB, so an escape hatch was built around the 16-KB limit. A file that is between 16 KB and 32 KB uses not one directory entry, but two. The first entry holds the first 16 blocks; the second entry holds the next 16 blocks. Beyond 32 KB, a third directory entry is used, and so on. The *Extent* field keeps track of the order of the directory entries so the system knows which 16 KB comes first, which comes second, and so on.

After an open call, the addresses of all the disk blocks are known, making read straightforward. The write call is also simple. It just requires allocating a free block from the bitmap in memory and then writing the block. Consecutive blocks on a file are not placed in consecutive blocks on the disk because the 8080 cannot process an interrupt and start reading the next block on time. Instead, interleaving is used to allow several blocks to be read on a single rotation.

CP/M is clearly not the last word in advanced file systems, but it is simple, fast, and can be implemented by a competent programmer in less than a week. For many embedded applications, it may be all that is needed.

6.4.3 The MS-DOS File System

To a first approximation, MS-DOS is a bigger and better version of CP/M. It runs only on Intel platforms, does not support multiprogramming, and runs only in the PC's real mode (which was originally the only mode). The shell has more features and there are more system calls, but the basic function of the operating system is still loading programs, handling the keyboard and screen, and managing the file system. It is the latter functionality that interests us here.

Spotify USA Inc. Exhibit 1009 Page 150

The MS-DOS file system was patterned closely on the CP/M file system, including the use of 8 + 3 (upper case) character file names. The first version (MS-DOS 1.0) was even limited to a single directory, like CP/M. However, starting with MS-DOS 2.0, the file system functionality was greatly expanded. The biggest improvement was the inclusion of a hierarchical file system in which directories could be nested to an arbitrary depth. This meant that the root directory (which still had a fixed maximum size) could contain subdirectories, and these could contain further subdirectories ad infinitem. Links in the style of UNIX were not permitted, so the file system formed a tree starting at the root directory.

It is common for different application programs to start out by creating a subdirectory in the root directory and put all their files there (or in subdirectories thereof), so that different applications do not conflict. Since directories are themselves just stored as files, there are no limits on the number of directories or files that may be created. Unlike CP/M, however, there is no concept of different users in MS-DOS. Consequently, the logged in user has access to all files.

To read a file, an MS-DOS program must first make an open system call to get a handle for it. The open system call specifies a path, which may be either absolute or relative to the current working directory. The path is looked up component by component until the final directory is located and read into memory. It is then searched for the file to be opened.

Although MS-DOS directories are variable sized, like CP/M, they use a fixed-size 32-byte directory entry. The format of an MS-DOS directory entry is shown in Fig. 6-32. It contains the file name, attributes, creation date and time, starting block, and exact file size. File names shorter than 8 + 3 characters are left justified and padded with spaces on the right, in each field separately. The *Attributes* field is new and contains bits to indicate that a file is read-only, needs to be archived, is hidden, or is a system file. Read-only files cannot be written. This is to protect them from accidental damage. The archived bit has no actual operating system function (i.e., MS-DOS does not examine or set it). The intention is to allow user-level archive programs to clear it upon archiving a file and to have other programs set it when modifying a file. In this way, a backup program can just examine this attribute bit on every file to see which files to back up. The hidden bit can be set to prevent a file from appearing in directory listings. Its main use is to avoid confusing novice users with files they might not understand. Finally, the system bit also hides files. In addition, system files cannot accidentally be deleted using the *del* command. The main components of MS-DOS have this bit set.

The directory entry also contains the date and time the file was created or last modified. The time is accurate only to ±2 sec because it is stored in a 2-byte field, which can store only 65,536 unique values (a day contains 86,400 unique seconds). The time field is subdivided into seconds (5 bits), minutes (6 bits), and hours (5 bits). The date counts in days using three subfields: day (5 bits), month (4 bits), and year–1980 (7 bits). With a 7-bit number for the year and time

Spotify USA Inc. Exhibit 1009 Page 151

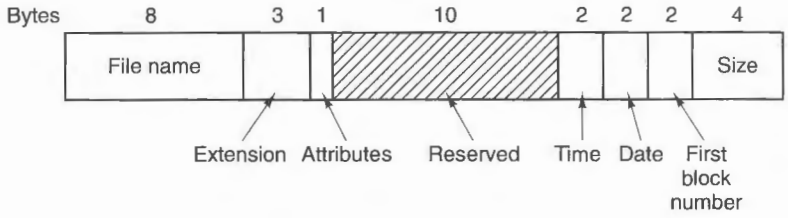


Figure 6-32. The MS-DOS directory entry.

beginning in 1980, the highest expressible year is 2107. Thus MS-DOS has a built-in Y2108 problem. To avoid catastrophe, MS-DOS users should begin with Y2108 compliance as early as possible. If MS-DOS had used the combined date and time fields as a 32-bit seconds counter, it could have represented every second exactly and delayed the catastrophe until 2116.

Unlike CP/M, which does not store the exact file size, MS-DOS does. Since a 32-bit number is used for the file size, in theory files can be as large as 4 GB. However, other limits (described below) restrict the maximum file size to 2 GB or less. A surprising large part of the entry (10 bytes) is unused.

Another way in which MS-DOS differs from CP/M is that it does not store a file's disk addresses in its directory entry, probably because the designers realized that large hard disks (by then common on minicomputers) would some day reach the MS-DOS world. Instead, MS-DOS keeps track of file blocks via a file allocation table in main memory. The directory entry contains the number of the first file block. This number is used as an index into a 64K entry FAT in main memory. By following the chain, all the blocks can be found. The operation of the FAT is illustrated in Fig. 6-14.

The FAT file system comes in three versions for MS-DOS: FAT-12, FAT-16, and FAT-32, depending on how many bits a disk address contains. Actually, FAT-32 is something of a misnomer since only the low-order 28 bits of the disk addresses are used. It should have been called FAT-28, but powers of two sound so much neater.

For all FATs, the disk block can be set to some multiple of 512 bytes (possibly different for each partition), with the set of allowed block sizes (called **cluster sizes** by Microsoft) being different for each variant. The first version of MS-DOS used FAT-12 with 512-byte blocks, giving a maximum partition size of $2^{12} \times 512$ bytes (actually only 4086×512 bytes because 10 of the disk addresses were used as special markers, such as end of file, bad block, etc. With these parameters, the maximum disk partition size was about 2 MB and the size of the FAT table in memory was 4096 entries of 2 bytes each. Using a 12-bit table entry would have been too slow.

This system worked well for floppy disks, but when hard disks came out, it became a problem. Microsoft solved the problem by allowing additional block

Spotify USA Inc. Exhibit 1009 Page 152

sizes of 1 KB, 2 KB, and 4 KB. This change preserved the structure and size of the FAT-12 table, but allowed disk partitions of up to 16 MB.

Since MS-DOS supported four disk partitions per disk drive, the new FAT-12 file system worked up to 64-MB disks. Beyond that, something had to give. What happened was the introduction of FAT-16, with 16-bit disk pointers. Additionally, block sizes of 8 KB, 16 KB, and 32 KB were permitted. (32,768 is the largest power of two that can be represented in 16 bits.) The FAT-16 table now occupied 128 KB of main memory all the time, but with the larger memories by then available, it was widely used and rapidly replaced the FAT-12 file system. The largest disk partition that can be supported by FAT-16 is 2 GB (64K entries of 32 KB each) and the largest disk 8 GB, namely four partitions of 2 GB each.

For business letters, this limit is not a problem, but for storing digital video using the DV standard, a 2-GB file holds just over 9 minutes of video. As a consequence of the fact that a PC disk can support only four partitions, the largest video that can be stored on a disk is about 38 minutes, no matter how large the disk is. This limit also means that the largest video that can be edited on line is less than 19 minutes, since both input and output files are needed.

Starting with the second release of Windows 95, the FAT-32 file system, with its 28-bit disk addresses, was introduced and the version of MS-DOS underlying Windows 95 was adapted to support FAT-32. In this system, partitions could theoretically be $2^{28} \times 2^{15}$ bytes, but they are actually limited to 2 TB (2048 GB) because internally the system keeps track of partition sizes in 512-byte sectors using a 32-bit number and $2^9 \times 2^{32}$ is 2 TB. The maximum partition size for various block sizes and all three FAT types is shown in Fig. 6-33.

Block size	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

Figure 6-33. Maximum partition size for different block sizes. The empty boxes represent forbidden combinations.

In addition to supporting larger disks, the FAT-32 file system has two other advantages over FAT-16. First, an 8-GB disk using FAT-32 can be a single partition. Using FAT-16 it has to be four partitions, which appears to the Windows user as the C:, D:, E:, and F: logical disk drives. It is up to the user to decide which file to place on which drive and keep track of what is where.

Spotify USA Inc. Exhibit 1009 Page 153

The other advantage of FAT-32 over FAT-16 is that for a given size disk partition, a smaller block size can be used. For example, for a 2-GB disk partition, FAT-16 must use 32-KB blocks, otherwise with only 64K available disk addresses, it cannot cover the whole partition. In contrast, FAT-32 can use, for example, 4-KB blocks for a 2-GB disk partition. The advantage of the smaller block size is that most files are much shorter than 32 KB. If the block size is 32 KB, a file of 10 bytes ties up 32 KB of disk space. If the average file is, say, 8 KB, then with a 32-KB block, ¾ of the disk will be wasted, not a terribly efficient way to use the disk. With an 8-KB file and a 4-KB block, there is no disk wastage, but the price paid is more RAM eaten up by the FAT. With a 4-KB block and a 2-GB disk partition, there are 512K blocks, so the FAT must have 512K entries in memory (occupying 2 MB of RAM).

MS-DOS uses the FAT to keep track of free disk blocks. Any block that is not currently allocated is marked with a special code. When MS-DOS needs a new disk block, it searches the FAT for an entry containing this code. Thus no bitmap or free list is required.

6.4.4 The Windows 98 File System

The original release of Windows 95 used the MS-DOS file system, including the use of 8 + 3 character file names and the FAT-12 and FAT-16 file systems. Starting with the second release of Windows 95, file names longer than 8 + 3 characters were permitted. In addition, FAT-32 was introduced, primarily to allow larger disk partitions larger than 2 GB and disks larger than 8 GB, which were then available. Both the long file names and FAT-32 were used in Windows 98 in the same form as in the second release of Windows 95. Below we will describe these features of the Windows 98 file system, which have been carried forward into Windows Me as well.

Since long file names are more exciting for users than the FAT structure, let us look at them first. One way to introduce long file names would have been to just invent a new directory structure. The problem with this approach is that if Microsoft had done this, people who were still in the process of converting from Windows 3 to Windows 95 or Windows 98 could not have accessed their files from both systems. A political decision was made within Microsoft that files created using Windows 98 must be accessible from Windows 3 as well (for dual-boot machines). This constraint forced a design for handling long file names that was backward compatible with the old MS-DOS 8 + 3 naming system. Since such backward compatibility constraints are not unusual in the computer industry, it is worth looking in detail at how Microsoft accomplished this goal.

The effect of this decision to be backward compatible meant that the Windows 98 directory structure had to be compatible with the MS-DOS directory structure. As we saw, this structure is just a list of 32-byte entries as shown in

Spotify USA Inc. Exhibit 1009 Page 154

Fig. 6-32. This format came directly from CP/M (which was written for the 8080), which goes to show how long (obsolete) structures can live in the computer world.

However, it was possible to now allocate the 10 unused bytes in the entries of Fig. 6-32, and that was done, as shown in Fig. 6-34. This change has nothing to do with long names, but it is used in Windows 98, so it is worth understanding.

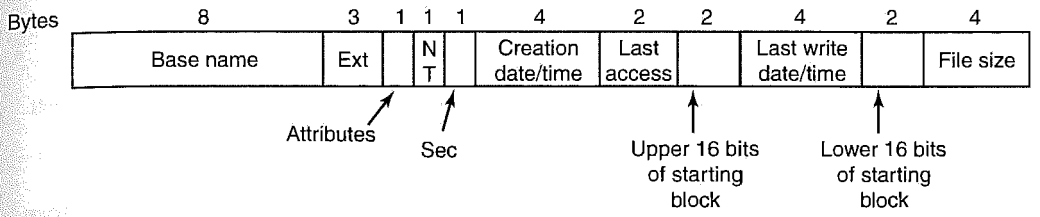


Figure 6-34. The extended MOS-DOS directory entry used in Windows 98.

The changes consist of the addition of five new fields where the 10 unused bytes used to be. The *NT* field is mostly there for some compatibility with Windows NT in terms of displaying file names in the correct case (in MS-DOS, all file names are upper case). The *Sec* field solves the problem that it is not possible to store the time of day in a 16-bit field. It provides additional bits so that the new *Creation time* field is accurate to 10 msec. Another new field is *Last access*, which stores the date (but not time) of the last access to the file. Finally, going to the FAT-32 file system means that block numbers are now 32 bits, so an additional 16-bit field is needed to store the upper 16 bits of the starting block number.

Now we come to the heart of the Windows 98 file system: how long file names are represented in a way that is backward compatible with MS-DOS. The solution chosen was to assign two names to each file: a (potentially) long file name (in Unicode, for compatibility with Windows NT), and an 8 + 3 name for compatibility with MS-DOS. Files can be accessed by either name. When a file is created whose name does not obey the MS-DOS naming rules (8 + 3 length, no Unicode, limited character set, no spaces, etc.), Windows 98 invents an MS-DOS name for it according to a certain algorithm. The basic idea is to take the first six characters of the name, convert them to upper case, if need be, and append ~1 to form the base name. If this name already exists, then the suffix ~2 is used, and so on. In addition, spaces and extra periods are deleted and certain special characters are converted to underscores. As an example, a file named *The time has come the walrus said* is assigned the MS-DOS name *THETIM~1*. If a subsequent file is created with the name *The time has come the rabbit said*, it is assigned the MS-DOS name *THETIM~2*, and so on.

Every file has an MS-DOS file name stored using the directory format of Fig. 6-34. If a file also has a long name, that name is stored in one or more directory entries directly preceding the MS-DOS file name. Each long-name entry holds up to 13 (Unicode) characters. The entries are stored in reverse order, with

Spotify USA Inc. Exhibit 1009 Page 155

the start of the file name just ahead of the MS-DOS entry and subsequent pieces before it. The format of each long-name entry is given in Fig. 6-35.

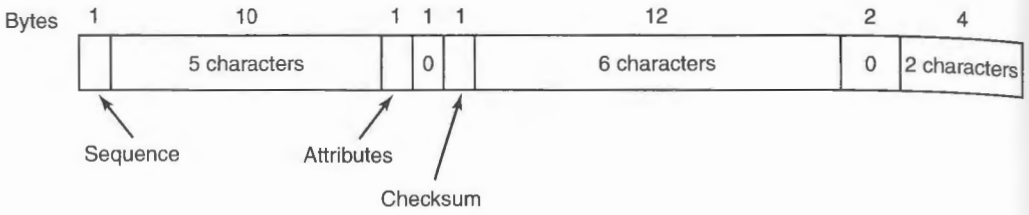


Figure 6-35. An entry for (part of) a long file name in Windows 98.

An obvious question is: "How does Windows 98 know whether a directory entry contains an MS-DOS file name or a (piece of a) long file name?" The answer lies in the *Attributes* field. For a long-name entry, this field has the value 0x0F, which represents an otherwise impossible combination of attributes. Old MS-DOS programs that read directories will just ignore it as invalid. Little do they know. The pieces of the name are sequenced using the first byte of the entry. The last part of the long name (the first entry in the sequence) is marked by adding 64 to the sequence number. Since only 6 bits are used for the sequence number, the theoretical maximum for file names is 63×13 or 819 characters. In fact they are limited to 260 characters for historical reasons.

Each long-name entry contains a *Checksum* field to avoid the following problem. First, a Windows 98 program creates a file with a long name. Second, the computer is rebooted to run MS-DOS or Windows 3. Third, an old program there then removes the MS-DOS file name from the directory but does not remove the long file name preceding it (because it does not know about it). Finally, some program creates a new file that reuses the newly-freed directory entry. At this point we have a valid sequence of long-name entries preceeding an MS-DOS file entry that has nothing to do with that file. The *Checksum* field allows Windows 98 to detect this situation by verifying that the MS-DOS file name following a long name does, in fact, belong to it. Of course, with only 1 byte being used, there is one chance in 256 that Windows 98 will not notice the file substitution.

To see an example of how long names work, consider the example of Fig. 6-36. Here we have a file called *The quick brown fox jumps over the lazy dog*. At 42-characters, it certainly qualifies as a long file name. The MS-DOS name constructed from it is *THEQUI~1* and is stored in the last entry.

Some redundancy is built into the directory structure to help detect problems in the event that an old Windows 3 program has made a mess of the directory. The sequence number byte at the start of each entry is not strictly needed since the 0x40 bit marks the first one, but it provides additional redundancy, for example. Also, the *Low* field of Fig. 6-36 (the lower half of the starting cluster) is 0 in all entries but the last one, again to avoid having old programs misinterpret it and

Spotify USA Inc. Exhibit 1009 Page 156

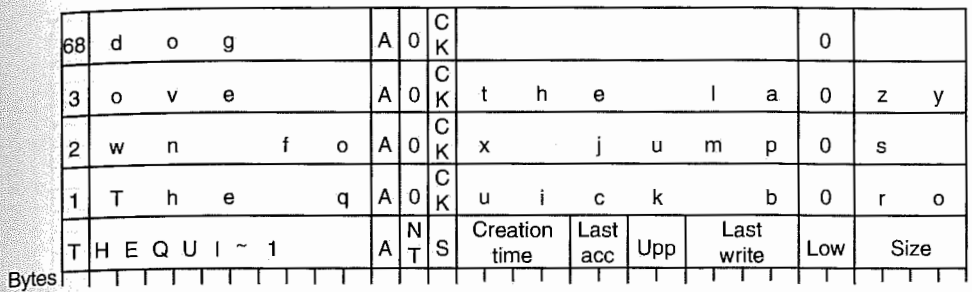


Figure 6-36. An example of how a long name is stored in Windows 98.

ruin the file system. The *NT* byte in Fig. 6-36 is used in NT and ignored in Windows 98. The *A* byte contains the attributes.

The implementation of the FAT-32 file system is conceptually similar to the implementation of the FAT-16 file system. However, instead of an array of 65,536 entries, there are as many entries as needed to cover the part of the disk with data on it. If the first million blocks are used, the table conceptually has 1 million entries. To avoid having all of them in memory at once, Windows 98 maintains a window into the table and keeps only in parts of it in memory at once.

6.4.5 The UNIX V7 File System

Even early versions of UNIX had a fairly sophisticated multiuser file system since it was derived from MULTICS. Below we will discuss the V7 file system, the one for the PDP-11 that made UNIX famous. We will examine modern versions in Chap. 10.

The file system is in the form of a tree starting at the root directory, with the addition of links, forming a directed acyclic graph. File names are up to 14 characters and can contain any ASCII characters except / (because that is the separator between components in a path) and NUL (because that is used to pad out names shorter than 14 characters). NUL has the numerical value of 0.

A UNIX directory entry contains one entry for each file in that directory. Each entry is extremely simple because UNIX uses the i-node scheme illustrated in Fig. 6-15. A directory entry contains only two fields: the file name (14 bytes) and the number of the i-node for that file (2 bytes), as shown in Fig. 6-37. These parameters limit the number of files per file system to 64K.

Like the i-node of Fig. 6-15, the UNIX i-nodes contains some attributes. The attributes contain the file size, three times (creation, last access, and last modification), owner, group, protection information, and a count of the number of directory entries that point to the i-node. The latter field is needed due to links. Whenever a new link is made to an i-node, the count in the i-node is increased. When a

Spotify USA Inc. Exhibit 1009 Page 157

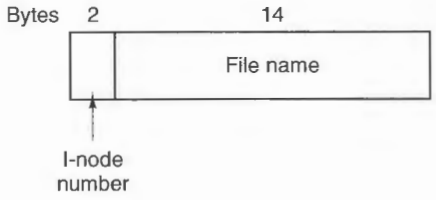


Figure 6-37. A UNIX V7 directory entry.

link is removed, the count is decremented. When it gets to 0, the i-node is reclaimed and the disk blocks are put back in the free list.

Keeping track of disk blocks is done using a generalization of Fig. 6-15 in order to handle very large files. The first 10 disk addresses are stored in the i-node itself, so for small files, all the necessary information is right in the i-node, which is fetched from disk to main memory when the file is opened. For somewhat larger files, one of the addresses in the i-node is the address of a disk block called a **single indirect block**. This block contains additional disk addresses. If this still is not enough, another address in the i-node, called a **double indirect block**, contains the address of a block that contains a list of single indirect blocks. Each of these single indirect blocks points to a few hundred data blocks. If even this is not enough, a **triple indirect block** can also be used. The complete picture is given in Fig. 6-38.

When a file is opened, the file system must take the file name supplied and locate its disk blocks. Let us consider how the path name */usr/ast/mbox* is looked up. We will use UNIX as an example, but the algorithm is basically the same for all hierarchical directory systems. First the file system locates the root directory. In UNIX its i-node is located at a fixed place on the disk. From this i-node, it locates the root directory which can be anywhere on the disk, but say block 1 in this case.

Then it reads the root directory and looks up the first component of the path, *usr*, in the root directory to find the i-node number of the file */usr*. Locating an i-node from its number is straightforward, since each one has a fixed location on the disk. From this i-node, the system locates the directory for */usr* and looks up the next component, *ast*, in it. When it has found the entry for *ast*, it has the i-node for the directory */usr/ast*. From this i-node it can find the directory itself and look up *mbox*. The i-node for this file is then read into memory and kept there until the file is closed. The lookup process is illustrated in Fig. 6-39.

Relative path names are looked up the same way as absolute ones, only starting from the working directory instead of starting from the root directory. Every directory has entries for . and .. which are put there when the directory is created. The entry . has the i-node number for the current directory, and the entry for .. has the i-node number for the parent directory. Thus, a procedure looking up *../dick/prog.c* simply looks up .. in the working directory, finds the i-node number

Spotify USA Inc. Exhibit 1009 Page 158

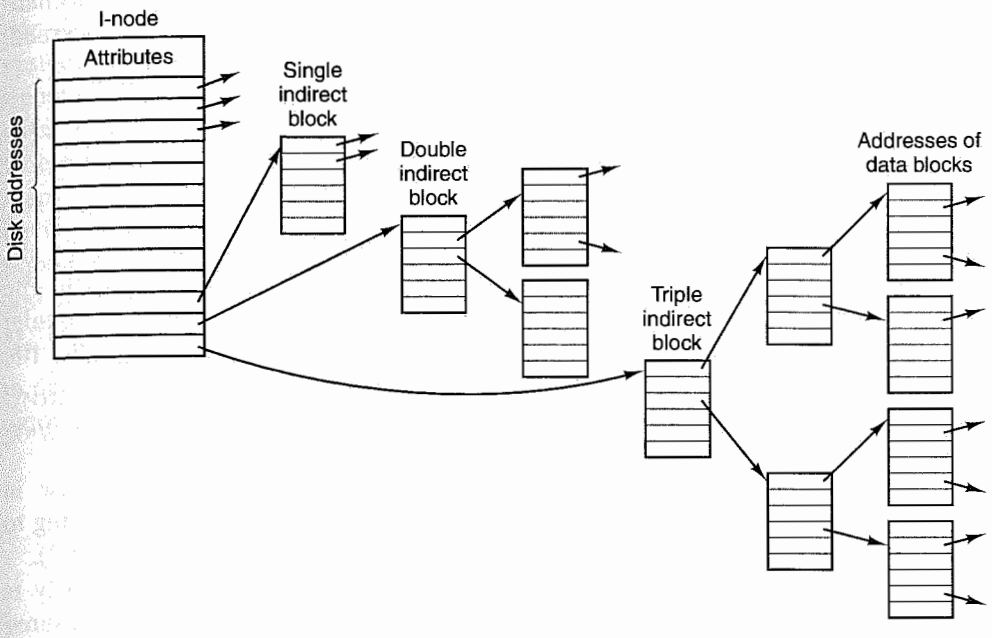


Figure 6-38. A UNIX i-node.

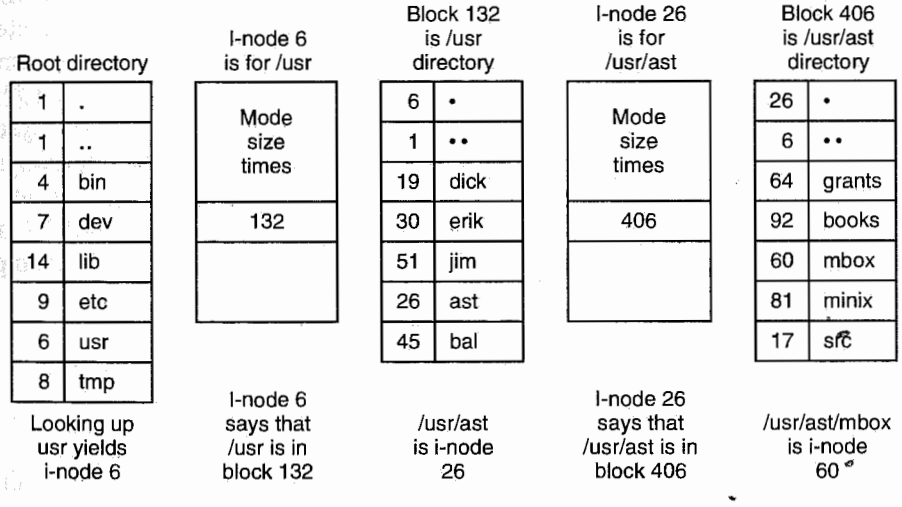


Figure 6-39. The steps in looking up */usr/ast/mbox*.

Spotify USA Inc. Exhibit 1009 Page 159

for the parent directory, and searches that directory for *dick*. No special mechanism is needed to handle these names. As far as the directory system is concerned, they are just ordinary ASCII strings, just the same as any other names.

6.5 RESEARCH ON FILE SYSTEMS

File systems have always attracted more research than other parts of the operating system and that is still the case. Some of this research relates to file system structure. Log-structured file systems and related topics are popular (Matthews et al., 1997; and Wang et al., 1999). The logical disk splits the file system into two distinct layers: the file system and the disk system (De Jonge et al., 1993). Building a file system from stackable layers is also a research topic (Heidemann and Popek, 1994).

Extensible file systems are analogous to the extensible kernels we saw in Chap 1. They allow new features to be added to the file system without having to redesign it from scratch (Karpovich et al., 1994; and Khalidi and Nelson, 1993).

Another research topic that has achieved a certain amount of popularity is making measurements of file system contents and use. People have measured what the file size distribution is, how long files live, whether all files are accessed equally, how reads compare to writes, and many other parameters (Douceur and Bolosky, 1999; Gill et al., 1994; Roselli and Lorch, 2000; and Vogels, 1999).

Other researchers have looked at file system performance and how to improve it using prefetching, caching, less copying, and other techniques. Usually, these researchers make measurements, find out where the bottlenecks are, eliminate at least one of them, and then make the measurements on the improved system to validate their results (Cao et al., 1995; Pai et al., 2000; and Patterson et al., 1995).

An area that many people rarely think about until disaster strikes is file system backup and recovery. Here, too, some new ideas have come up for better ways to do things (Chen et al., 1996; Devarakonda et al., 1996; and Hutchinson et al., 1999). Somewhat related to this is the question of what to do when a user removes a file—remove it or hide it? The Elephant file system, for example, never forgets (Santry et al., 1999a; and Santry et al., 1999b).

6.6 SUMMARY

When seen from the outside, a file system is a collection of files and directories, plus operations on them. Files can be read and written, directories can be created and destroyed, and files can be moved from directory to directory. Most modern file systems support a hierarchical directory system, in which directories may have subdirectories and these may have subsubdirectories ad infinitum.

Spotify USA Inc. Exhibit 1009 Page 160

When seen from the inside, a file system looks quite different. The file system designers have to be concerned with how storage is allocated, and how the system keeps track of which block goes with which file. Possibilities include contiguous files, linked lists, file allocation tables, and i-nodes. Different systems have different directory structures. Attributes can go in the directories or somewhere else (e.g., an i-node). Disk space can be managed using free lists of bitmaps. File system reliability is enhanced by making incremental dumps and by having a program that can repair sick file systems. File system performance is important and can be enhanced in several ways, including caching, read ahead, and carefully placing the blocks of a file close to each other. Log-structured file systems also improve performance by doing writes in large units.

Examples of file systems include ISO 9660, CP/M, MS-DOS, Windows 98, and UNIX. These differ in many ways, including how they keep track of which blocks go with which file, directory structure, and management of free disk space.

PROBLEMS

1. Give 5 different path names for the file */etc/passwd*. *Hint*: Think about the directory entries "." and "..".
2. In Windows, when a user double clicks on a file listed by Windows Explorer, a program is run and given that file as a parameter. List two different ways the operating system could know which program to run?
3. In early UNIX systems, executable files (*a.out* files) began with a very specific magic number, not one chosen at random. These files began with a header, followed by the text and data segments. Why do you think a very specific number was chosen for executable files, whereas other file types had a more-or-less random magic number as the first word?
4. In Fig. 6-4, one of the attributes is the record length. Why does the operating system ever care about this?
5. Is the open system call in UNIX absolutely essential? What would the consequences be of not having it?
6. Systems that support sequential files always have an operation to rewind files. Do systems that support random access files need this too?
7. Some operating systems provide a system call rename to give a file a new name. Is there any difference at all between using this call to rename a file, and just copying the file to a new file with the new name, followed by deleting the old one?
8. In some systems it is possible to map part of a file into memory. What restrictions must such systems impose? How is this partial mapping implemented?

Spotify USA Inc. Exhibit 1009 Page 161

9. A simple operating system only supports a single directory but allows that directory to have arbitrarily many files with arbitrarily long file names. Can something approximating a hierarchical file system be simulated? How?

10. In UNIX and Windows, random access is done by having a special system call that moves the "current position" pointer associated with a file to a given byte in the file. Propose an alternative way to do random access without having this system call.

11. Consider the directory tree of Fig. 6-10. If */usr/jim* is the working directory, what is the absolute path name for the file whose relative path name is *../ast/x*?

12. Contiguous allocation of files leads to disk fragmentation, as mentioned in the text, because some space in the last disk block will be wasted in files whose length is not an integral number of blocks. Is this internal fragmentation or external fragmentation? Make an analogy with something discussed in the previous chapter.

13. One way to use contiguous allocation of the disk and not suffer from holes is to compact the disk every time a file is removed. Since all files are contiguous, copying a file requires a seek and rotational delay to read the file, followed by the transfer at full speed. Writing the file back requires the same work. Assuming a seek time of 5 msec, a rotational delay of 4 msec, a transfer rate of 8 MB/sec, and an average file size of 8 KB, how long does it take to read a file into main memory then write it back to the disk at a new location? Using these numbers, how long would it take to compact half of a 16-GB disk?

14. In light of the answer to the previous question, does compacting the disk ever make any sense?

15. Some digital consumer devices need to store data, for example as files. Name a modern device that requires file storage and for which contiguous allocation would be a fine idea.

16. How does MS-DOS implement random access to files?

17. Consider the i-node shown in Fig. 6-15. If it contains 10 direct addresses of 4 bytes each and all disk blocks are 1024 KB, what is the largest possible file?

18. It has been suggested that efficiency could be improved and disk space saved by storing the data of a short file within the i-node. For the i-node of Fig. 6-15, how many bytes of data could be stored inside the i-node?

19. Two computer science students, Carolyn and Elinor, are having a discussion about i-nodes. Carolyn maintains that memories have gotten so large and so cheap that when a file is opened, it is simpler and faster just to fetch a new copy of the i-node into the i-node table, rather than search the entire table to see if it is already there. Elinor disagrees. Who is right?

20. Name one advantage of hard links over symbolic links and one advantage of symbolic links over hard links.

21. Free disk space can be kept track of using a free list or a bitmap. Disk addresses require D bits. For a disk with B blocks, F of which are free, state the condition under which the free list uses less space than the bitmap. For D having the value 16 bits, express your answer as a percentage of the disk space that must be free.

Spotify USA Inc. Exhibit 1009 Page 162

22. The beginning of a free space bitmap looks like this after the disk partition is first formatted: 1000 0000 0000 0000 (the first block is used by the root directory). The system always searches for free blocks starting at the lowest numbered block, so after writing file *A*, which uses 6 blocks, the bitmap looks like this: 1111 1110 0000 0000. Show the bitmap after each of the following additional actions:

(a) File *B* is written, using 5 blocks
(b) File *A* is deleted
(c) File *C* is written, using 8 blocks
(d) File *B* is deleted.

23. What would happen if the bitmap or free list containing the information about free disk blocks was completely lost due to a crash? Is there any way to recover from this disaster, or is it bye-bye disk? Discuss your answer for UNIX and the FAT-16 file system separately.

24. Oliver Owl's night job at the university computing center is to change the tapes used for overnight data backups. While waiting for each tape to complete, he works on writing his thesis that proves Shakespeare's plays were written by extraterrestrial visitors. His text processor runs on the system being backed up since that is the only one they have. Is there a problem with this arrangement?

25. We discussed making incremental dumps in some detail in the text. In Windows it is easy to tell when to dump a file because every file has an archive bit. This bit is missing in UNIX. How do UNIX backup programs know which files to dump?

26. Suppose that file 21 in Fig. 6-24 was not modified since the last dump. In what way would the four bitmaps of Fig. 6-25 be different?

27. It has been suggested that the first part of each UNIX file be kept in the same disk block as its i-node. What good would this do?

28. Consider Fig. 6-26. Is it possible that for some particular block number the counter in *both* lists have the value 2? How should this problem be corrected?

29. The performance of a file system depends upon the cache hit rate (fraction of blocks found in the cache). If it takes 1 msec to satisfy a request from the cache, but 40 msec to satisfy a request if a disk read is needed, give a formula for the mean time required to satisfy a request if the hit rate is h. Plot this function for values of h from 0 to 1.0.

30. A floppy disk has 40 cylinders. A seek takes 6 msec per cylinder moved. If no attempt is made to put the blocks of a file close to each other, two blocks that are logically consecutive (i.e., follow one another in the file) will be about 13 cylinders apart, on the average. If, however, the operating system makes an attempt to cluster related blocks, the mean interblock distance can be reduced to 2 cylinders (for example). How long does it take to read a 100-block file in both cases, if the rotational latency is 100 msec and the transfer time is 25 msec per block?

31. Consider the idea behind Fig. 6-20, but now for a disk with a mean seek time of 8 msec, a rotational rate of 15,000 rpm, and 262,144 bytes per track. What are the data rates for block sizes of 1 KB, 2 KB, and 4 KB, respectively?

Spotify USA Inc. Exhibit 1009 Page 163

32. A certain file system uses 2-KB disk blocks. The median file size is 1 KB. If all files were exactly 1 KB, what fraction of the disk space would be wasted? Do you think the wastage for a real file system will be higher than this number or lower than it? Explain your answer.

33. CP/M was designed to run with a small floppy disk as the mass storage device. Suppose that it were to be ported to a modern computer with a large hard disk. What is the largest possible disk that could be supported without changing the size of the directory format shown in Fig. 6-31? The fields within the directory and other system parameters may be changed if necessary. What changes would you make?

34. The MS-DOS FAT-16 table contains 64K entries. Suppose that one of the bits had been needed for some other purpose and that the table contained exactly 32,768 entries instead. With no other changes, what would the largest MS-DOS file have been under this conditions?

35. Files in MS-DOS have to compete for space in the FAT-16 table in memory. If one file uses k entries, that is k entries that are not available to any other file, what constraint does this place on the total length of all files combined?

36. A UNIX file system has 1-KB blocks and 4-byte disk addresses. What is the maximum file size if i-nodes contain 10 direct entries, and one single, double, and triple indirect entry each?

37. How many disk operations are needed to fetch the i-node for the file */usr/ast/courses/os/handout.t*? Assume that the i-node for the root directory is in memory, but nothing else along the path is in memory. Also assume that all directories fit in one disk block.

38. In many UNIX systems, the i-nodes are kept at the start of the disk. An alternative design is to allocate an i-node when a file is created and put the i-node at the start of the first block of the file. Discuss the pros and cons of this alternative.

39. Write a program that reverses the bytes of a file, so that the last byte is now first and the first byte is now last. It must work with an arbitrarily long file, but try to make it reasonably efficient.

40. Write a program that starts at a given directory and descends the file tree from that point recording the sizes of all the files it finds. When it is all done, it should print a histogram of the file sizes using a bin width specified as a parameter (e.g., with 1024, file sizes of 0 to 1023 go in one bin, 1024 to 2047 go in the next bin, etc.).

41. Write a program that scans all directories in a UNIX file system and finds and locates all i-nodes with a hard link count of two or more. For each such file, it lists together all file names that point to the file.

42. Write a new version of the UNIX *ls* program. This version takes as an argument one or more directory names and for each directory lists all the files in that directory, one line per file. Each field should be formatted in a reasonable way given its type. List only the first disk address, if any.

43. Write CP/M in C or C++. Look on the Web to find information about it.

Spotify USA Inc. Exhibit 1009 Page 164

7

MULTIMEDIA OPERATING SYSTEMS

Digital movies, video clips, and music are becoming an increasingly common way to present information and entertainment using a computer. Audio and video files can be stored on a disk and played back on demand. However, their characteristics are very different from the traditional text files that current file systems were designed for. As a consequence, new kinds of file systems are needed to handle them. Stronger yet, storing and playing back audio and video puts new demands on the scheduler and other parts of the operating system as well. In the sections that follow, we will study many of these issues and their implications for operating systems that are designed to handle multimedia.

Usually, digital movies go under the name **multimedia**, which literally means more than one medium. Under this definition, this book is a multimedia work. After all, it contains two media: text and images (the figures). However, most people use the term "multimedia" to mean a document containing two or more *continuous* media, that is media that must be played back over some time interval. In this book, we will use the term multimedia in this sense.

Another term that is somewhat ambiguous is "video." In a technical sense, it is just the image portion of a movie (as opposed to the sound portion). In fact, camcorders and televisions often have two connectors, one labeled "video" and one labeled "audio," since the signals are separate. However, the term "digital video" normally refers to the complete product, with both image and sound. Below we will use the term "movie" to refer to the complete product. Note that a movie in this sense need not be a two-hour long film produced by a Hollywood

Spotify USA Inc. Exhibit 1009 Page 165

studio at a cost exceeding that of a Boeing 747. A 30-sec news clip downloaded from CNN's home page over the Internet is also a movie under our definition. We will also call these "video clips" when we are referring to very short movies.

7.1 INTRODUCTION TO MULTIMEDIA

Before getting into the technology of multimedia, a few words about its current and future uses are perhaps helpful to set the stage. On a single computer, multimedia often means playing a prerecorded movie from a **DVD** (**Digital Versatile Disk**). DVDs are optical disks that use the same 120-mm polycarbonate (plastic) blanks that CD-ROMs use, but are recorded at a higher density, giving a capacity of between 5 GB and 17 GB, depending on the format.

Another use of multimedia is for downloading video clips over the Internet. Many Web pages have items that can be clicked on to download short movies. At 56 Kbps, downloading even a short video clip takes a long time, but as faster distribution technologies take over, such as cable TV and **ADSL** (**Asymmetric Digital Subscriber Loop**), the presence of video clips on the Internet will skyrocket.

Another area in which multimedia must be supported is in the creation of videos themselves. Multimedia editing systems exist and for best performance need to run on an operating system that supports multimedia as well as traditional work.

Yet another arena where multimedia is becoming important is in computer games. Games often run short video clips to depict some kind of action. The clips are usually short, but there are many of them and the correct one is selected dynamically, depending on some action the user has taken. These are increasingly sophisticated

Finally, the holy grail of the multimedia world is **video on demand**, by which people mean the ability for consumers at home to select a movie using their television remote control (or mouse) and have it displayed on their TV set (or computer monitor) on the spot. To enable video on demand, a special infrastructure is needed. In Fig. 7-1 we see two possible video-on-demand infrastructures. Each one contains three essential components: one or more video servers, a distribution network, and a set-top box in each house for decoding the signal. The **video server** is a powerful computer that stores many movies in its file system and plays them back on demand. Sometimes mainframes are used as video servers, since connecting, say, 1000 large disks to a mainframe is straightforward, whereas connecting 1000 disks of any kind to a personal computer is a serious problem. Much of the material in the following sections is about video servers and their operating systems.

The distribution network between the user and the video server must be capable of transmitting data at high speed and in real time. The design of such networks is interesting and complex, but falls outside the scope of this book. We

Spotify USA Inc. Exhibit 1009 Page 166

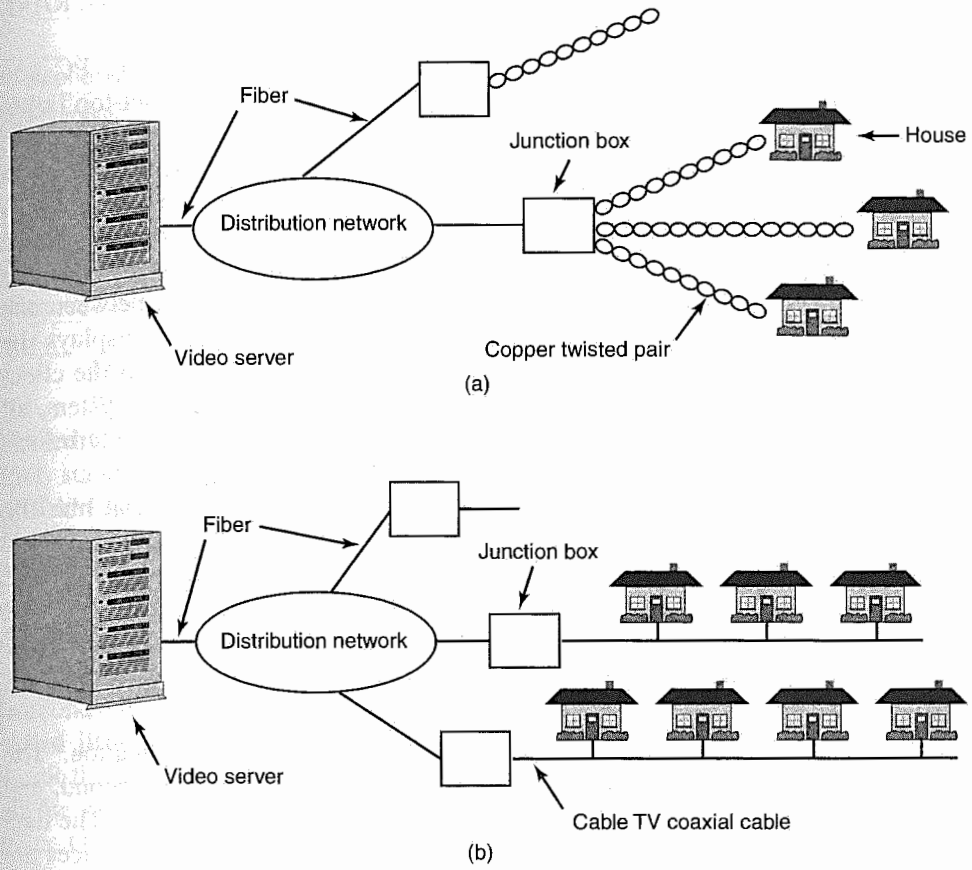


Figure 7-1. Video on demand using different local distribution technologies. (a) ADSL. (b) Cable TV.

will not say any more about them except to note that these networks always use fiber optics from the video server down to a junction box in each neighborhood where customers live. In ADSL systems, which are provided by telephone companies, the existing twisted-pair telephone line provides the last kilometer or so of transmission. In cable TV systems, which are provided by cable operators, existing cable TV wiring is used for the local distribution. ADSL has the advantage of giving each user a dedicated channel, hence guaranteed bandwidth, but the bandwidth is low (a few megabits/sec) due to limitations of existing telephone wire. Cable TV uses high-bandwidth coaxial cable (at gigabits/sec), but many users have to share the same cable, giving contention for it and no guaranteed bandwidth to any individual user.

The last piece of the system is the **set-top box**, where the ADSL or TV cable terminates. This device is, in fact, a normal computer, with certain special chips

Spotify USA Inc. Exhibit 1009 Page 167

for video decoding and decompression. As a minimum, it contains a CPU, RAM, ROM, and interface to ADSL or the cable.

An alternative to a set-top box is to use the customer's existing PC and display the movie on the monitor. Interestingly enough, the reason set-top boxes are even considered given that most customers probably already have a computer, is that video-on-demand operators expect that people will want to watch movies in their living rooms, which usually contain a TV but rarely a computer. From a technical perspective, using a personal computer instead of a set-top box makes far more sense since it is more powerful, has a large disk, and has a far higher resolution display. Either way, we will often make a distinction between the video server and the client process at the user end that decodes and displays the movie. In terms of system design, however, it does not matter much if the client process runs on a set-top box or on a PC. For a desktop video editing system, all the processes run on the same machine, but we will continue to use the terminology of server and client to make it clear which process is doing what.

Getting back to multimedia itself, it has two key characteristics that must be well understood to deal with it successfully:

1. Multimedia uses extremely high data rates.

2. Multimedia requires real-time playback.

The high data rates come from the nature of visual and acoustic information. The eye and the ear can process prodigious amounts of information per second, and have to be fed at that rate to produce an acceptable viewing experience. The data rates of a few digital multimedia sources and some common hardware devices are listed in Fig. 7-2. We will discuss some of these encoding formats later in this chapter. What should be noted is the high data rates multimedia requires, the need for compression, and the amount of storage that is required. For example, an uncompressed 2-hour HDTV movie fills a 570-GB file. A video server that stores 1000 such movies needs 570 TB of disk space, a nontrivial amount by current standards. What is also of note is that without data compression, current hardware cannot keep up with the data rates produced. We will examine video compression later in this chapter.

The second demand that multimedia puts on a system is the need for real-time data delivery. The video portion of a digital movie consists of some number of frames per second. The NTSC system, used in North and South America and Japan, runs at 30 frames/sec (29.97 for the purist), whereas the PAL and SECAM systems, used in most of the rest of the world, runs at 25 frames/sec (25.00 for the purist). Frames must be delivered at precise intervals of ca. 33.3 msec or 40 msec, respectively, or the movie will look choppy.

Officially NTSC stands for National Television Standards Committee, but the poor way color was hacked into the standard when color television was invented

Spotify USA Inc. Exhibit 1009 Page 168

Source	Mbps	GB/hr
Telephone (PCM)	0.064	0.03
MP3 music	0.14	0.06
Audio CD	1.4	0.62
MPEG-2 movie (640 × 480)	4	1.76
Digital camcorder (720 × 480)	25	11
Uncompressed TV (640 × 480)	221	97
Uncompressed HDTV (1280 × 720)	648	288

Device	Mbps
Fast Ethernet	100
EIDE disk	133
ATM OC-3 network	156
SCSI UltraWide disk	320
IEEE 1394 (FireWire)	400
Gigabit Ethernet	1000
SCSI Ultra-160 disk	1280

Figure 7-2. Some data rates for multimedia and high-performance I/O devices. Note that 1 Mbps is 10^6 bits/sec but 1 GB is 2^{30} bytes.

has led to the industry joke that it really stands for Never Twice the Same Color. PAL stands for Phase Alternating Line. Technically it is the best of the systems. SECAM is used in France (and was intended to protect French TV manufacturers from foreign competition) and stands for SEquentiel Couleur Avec Memoire. SECAM is also used in Eastern Europe because when television was introduced there, the then-Communist governments wanted to keep everyone from watching German (PAL) television, so they chose an incompatible system.

The ear is more sensitive than the eye, so a variance of even a few milliseconds in delivery times will be noticeable. Variability in delivery rates is called **jitter** and must be strictly bounded for good performance. Note that jitter is not the same as delay. If the distribution network of Fig. 7-1 uniformly delays all the bits by exactly 5.000 sec, the movie will start slightly later, but will look fine. On the other hand, if it randomly delays frames by between 100 and 200 msec, the movie will look like an old Charlie Chaplin film, no matter who is starring.

The real-time properties required to play back multimedia acceptably are often described by **quality of service** parameters. They include average bandwidth available, peak bandwidth available, minimum and maximum delay (which together bound the jitter), and bit loss probability. For example, a network operator could offer a service guaranteeing an average bandwidth of 4 Mbps, 99% of the transmission delays in the interval 105 to 110 msec, and a bit loss rate of 10^{-10}, which would be fine for MPEG-2 movies. The operator could also offer a cheaper, lower-grade service, with an average bandwidth of 1 Mbps (e.g., ADSL), in which case the quality would have to be compromised somehow, possibly by lowering the resolution, dropping the frame rate, or discarding the color information and showing the movie in black and white.

The most common way to provide quality of service guarantees is to reserve capacity in advance for each new customer. The resources reserved include a portion of the CPU, memory buffers, disk transfer capacity, and network bandwidth. If a new customer comes along and wants to watch a movie, but the video server

Spotify USA Inc. Exhibit 1009 Page 169

or network calculates that it does not have sufficient capacity for another customer, it has to reject the new customer to avoid degrading the service to current customers. As a consequence, multimedia servers need resource reservation schemes and an **admission control algorithm** to decide when they can handle more work.

7.2 MULTIMEDIA FILES

In most systems, an ordinary text file consists of a linear sequence of bytes without any structure that the operating system knows about or cares about. With multimedia, the situation is more complicated. To start with, video and audio are completely different. They are captured by distinct devices (CCD chip versus microphone), have a different internal structure (video has 25-30 frames/sec; audio has 44,100 samples/sec), and they are played back by different devices (monitor versus loudspeakers).

Furthermore, most Hollywood movies are now aimed at a worldwide audience, most of which does not speak English. The latter point is dealt with in one of two ways. For some countries, an additional sound track is produced, with the voices dubbed into the local language (but not the sound effects). In Japan, all televisions have two sound channels to allow the viewer to listen to foreign films in either the original language or in Japanese. A button on the remote control is used for language selection. In still other countries, the original sound track is used, with subtitles in the local language.

In addition, many TV movies now provide closed-caption subtitles in English as well, to allow English-speaking but hearing-impaired people to watch the movie. The net result is that a digital movie may actually consist of many files: one video file, multiple audio files, and multiple text files with subtitles in various languages. DVDs have the capability for storing up to 32 language and subtitle files. A simple set of multimedia files is shown in Fig. 7-3. We will explain the meaning of fast forward and fast backward later in this chapter.

As a consequence, the file system needs to keep track of multiple "subfiles" per file. One possible scheme is to manage each subfile as a traditional file (e.g., using an i-node to keep track of its blocks) and to have a new data structure that lists all the subfiles per multimedia file. Another way is to invent a kind of two-dimensional i-node, with each column listing the blocks of each subfile. In general, the organization must be such that the viewer can dynamically choose which audio and subtitle tracks to use at the time the movie is viewed.

In all cases, some way to keep the subfiles synchronized is also needed so that when the selected audio track is played back it remains in sync with the video. If the audio and video get even slightly out of sync, the viewer may hear an actor's words before or after his lips move, which is easily detected and fairly annoying.

Spotify USA Inc. Exhibit 1009 Page 170

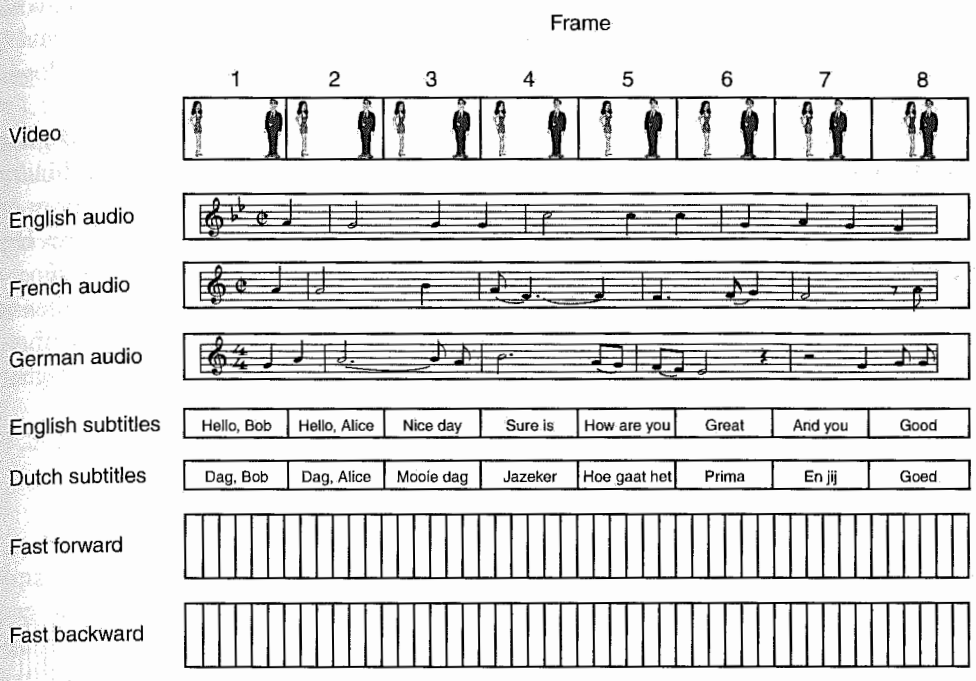


Figure 7-3. A movie may consist of several files.

To better understand how multimedia files are organized, it is necessary to understand how digital audio and video work in some detail. We will now give an introduction to these topics.

7.2.1 Audio Encoding

An audio (sound) wave is a one-dimensional acoustic (pressure) wave. When an acoustic wave enters the ear, the eardrum vibrates, causing the tiny bones of the inner ear to vibrate along with it, sending nerve pulses to the brain. These pulses are perceived as sound by the listener. In a similar way, when an acoustic wave strikes a microphone, the microphone generates an electrical signal, representing the sound amplitude as a function of time.

The frequency range of the human ear runs from 20 Hz to 20,000 Hz, although some animals, notably dogs, can hear higher frequencies. The ear hears logarithmically, so the ratio of two sounds with amplitudes A and B is conventionally expressed in **dB** (**decibels**) according to the formula

$$\text{dB} = 20 \log_{10}(A/B)$$

If we define the lower limit of audibility (a pressure of about 0.0003 dyne/cm^2) for a 1-kHz sine wave as 0 dB, an ordinary conversation is about 50 dB and the

Spotify USA Inc. Exhibit 1009 Page 171

pain threshold is about 120 dB, a dynamic range of a factor of 1 million. To avoid any confusion, *A* and *B* above are *amplitudes*. If we were to use the power level, which is proportional to the square of the amplitude, the coefficient of the logarithm would be 10, not 20.

Audio waves can be converted to digital form by an **ADC (Analog Digital Converter)**. An ADC takes an electrical voltage as input and generates a binary number as output. In Fig. 7-4(a) we see an example of a sine wave. To represent this signal digitally, we can sample it every ΔT seconds, as shown by the bar heights in Fig. 7-4(b). If a sound wave is not a pure sine wave, but a linear superposition of sine waves where the highest frequency component present is f, then it is sufficient to make samples at a frequency $2f$. This result was proven mathematically by H. Nyquist in 1924. Sampling more often is of no value since the higher frequencies that such sampling could detect are not present.

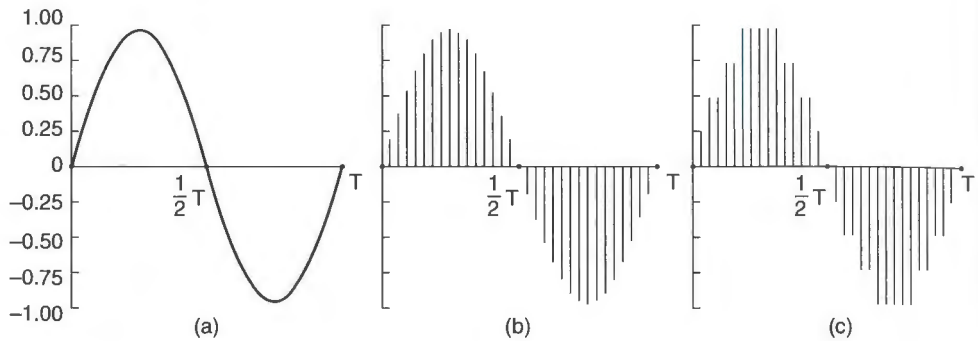


Figure 7-4. (a) A sine wave. (b) Sampling the sine wave. (c) Quantizing the samples to 4 bits.

Digital samples are never exact. The samples of Fig. 7-4(c) allow only nine values, from –1.00 to +1.00 in steps of 0.25. Consequently, 4 bits are needed to represent all of them. An 8-bit sample would allow 256 distinct values. A 16-bit sample would allow 65,536 distinct values. The error introduced by the finite number of bits per sample is called the **quantization noise**. If it is too large, the ear detects it.

Two well-known examples of sampled sound are the telephone and audio compact discs. **Pulse code modulation** is used within the telephone system and uses 7-bit (North America and Japan) or 8-bit (Europe) samples 8000 times per second. This system gives a data rate of 56,000 bps or 64,000 bps. With only 8000 samples/sec, frequencies above 4 kHz are lost.

Audio CDs are digital with a sampling rate of 44,100 samples/sec, enough to capture frequencies up to 22,050 Hz, which is good for people, bad for dogs. The samples are 16 bits each, and are linear over the range of amplitudes. Note that 16-bit samples allow only 65,536 distinct values, even though the dynamic range

Spotify USA Inc. Exhibit 1009 Page 172

of the ear is about 1 million when measured in steps of the smallest audible sound. Thus using only 16 bits per sample introduces some quantization noise (although the full dynamic range is not covered—CDs are not supposed to hurt). With 44,100 samples/sec of 16 bits each, an audio CD needs a bandwidth of 705.6 Kbps for monaural and 1.411 Mbps for stereo (see Fig. 7-2). Audio compression is possible based on psychoacoustic models of how human hearing works. A compression of 10x is possible using the MPEG layer 3 (MP3) system. Portable music players for this format have been common in recent years.

Digitized sound can easily be processed by computers in software. Dozens of programs exist for personal computers to allow users to record, display, edit, mix, and store sound waves from multiple sources. Virtually all professional sound recording and editing is digital nowadays.

7.2.2 Video Encoding

The human eye has the property that when an image is flashed on the retina, it is retained for some number of milliseconds before decaying. If a sequence of images is flashed at 50 or more images/sec, the eye does not notice that it is looking at discrete images. All video- and film-based motion picture systems exploit this principle to produce moving pictures.

To understand video systems, it is best to start with simple, old-fashioned black-and-white television. To represent the two-dimensional image in front of it as a one-dimensional voltage as a function of time, the camera scans an electron beam rapidly across the image and slowly down it, recording the light intensity as it goes. At the end of the scan, called a **frame**, the beam retraces. This intensity as a function of time is broadcast, and receivers repeat the scanning process to reconstruct the image. The scanning pattern used by both the camera and the receiver is shown in Fig. 7-5. (As an aside, CCD cameras integrate rather than scan, but some cameras and all CRT monitors do scan.)

The exact scanning parameters vary from country to country. NTSC has 525 scan lines, a horizontal to vertical aspect ratio of 4:3, and 30 frames/sec. The European PAL and SECAM systems have 625 scan lines, the same aspect ratio of 4:3, and 25 frames/sec. In both systems, the top few and bottom few lines are not displayed (to approximate a rectangular image on the original round CRTs). Only 483 of the 525 NTSC scan lines (and 576 of the 625 PAL/SECAM scan lines) are displayed.

While 25 frames/sec is enough to capture smooth motion, at that frame rate many people, especially older ones, will perceive the image to flicker (because the old image has faded off the retina before the new one appears). Rather than increase the frame rate, which would require using more scarce bandwidth, a different approach is taken. Instead of displaying the scan lines in order fromtop to bottom, first all the odd scan lines are displayed, then the even ones are displayed. Each of these half frames is called a **field**. Experiments have shown that although

Spotify USA Inc. Exhibit 1009 Page 173

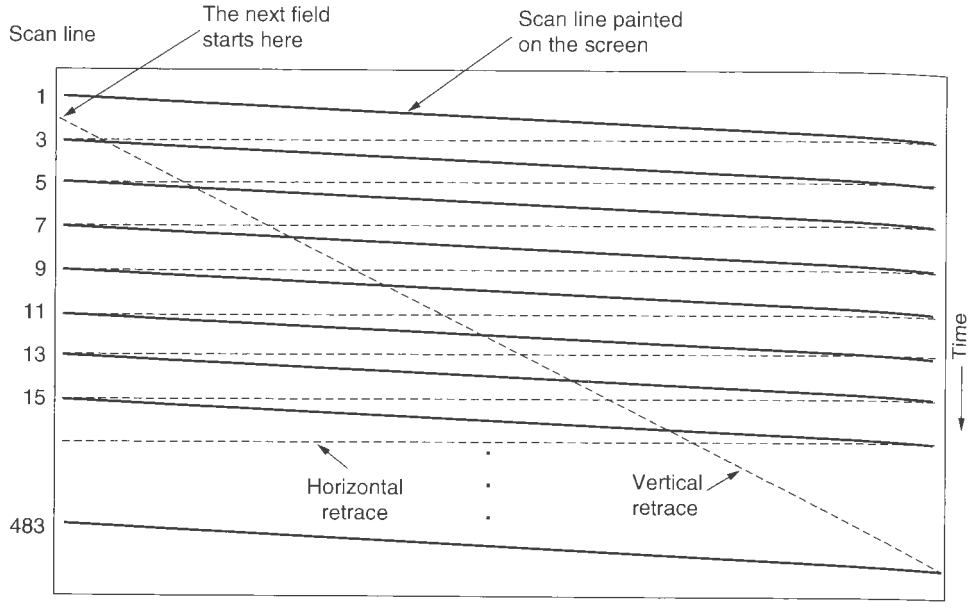


Figure 7-5. The scanning pattern used for NTSC video and television.

people notice flicker at 25 frames/sec, they do not notice it at 50 fields/sec. This technique is called **interlacing**. Noninterlaced television or video is said to be **progressive**.

Color video uses the same scanning pattern as monochrome (black and white), except that instead of displaying the image with one moving beam, three beams moving in unison are used. One beam is used for each of the three additive primary colors: red, green, and blue (RGB). This technique works because any color can be constructed from a linear superposition of red, green, and blue with the appropriate intensities. However, for transmission on a single channel, the three color signals must be combined into a single **composite** signal.

To allow color transmissions to be viewed on black-and-white receivers, all three systems linearly combine the RGB signals into a **luminance** (brightness) signal, and two **chrominance** (color) signals, although they all use different coefficients for constructing these signals from the RGB signals. Interestingly enough, the eye is much more sensitive to the luminance signal than to the chrominance signals, so the latter need not be transmitted as accurately. Consequently, the luminance signal can be broadcast at the same frequency as the old black-and-white signal, so it can be received on black-and-white television sets. The two chrominance signals are broadcast in narrow bands at higher frequencies. Some television sets have knobs or controls labeled brightness, hue, and saturation (or brightness, tint and color) for controlling these three signals separately.

Spotify USA Inc. Exhibit 1009 Page 174

Understanding luminance and chrominance is necessary for understanding how video compression works.

So far we have looked at analog video. Now let us turn to digital video. The simplest representation of digital video is a sequence of frames, each consisting of a rectangular grid of picture elements, or **pixels**. For color video, 8 bits per pixel are used for each of the RGB colors, giving 16 million colors, which is enough. The human eye cannot even distinguish this many colors, let alone more.

To produce smooth motion, digital video, like analog video, must display at least 25 frames/sec. However, since good quality computer monitors often rescan the screen from images stored in video RAM at 75 times per second or more, interlacing is not needed Consequently, all computer monitors use progressive scanning. Just repainting (i.e., redrawing) the same frame three times in a row is enough to eliminate flicker.

In other words, smoothness of motion is determined by the number of *different* images per second, whereas flicker is determined by the number of times the screen is painted per second. These two parameters are different. A still image painted at 20 frames/sec will not show jerky motion but it will flicker because one frame will decay from the retina before the next one appears. A movie with 20 different frames per second, each of which is painted four times in a row at 80 Hz, will not flicker, but the motion will appear jerky.

The significance of these two parameters becomes clear when we consider the bandwidth required for transmitting digital video over a network. Current computer monitors all use the 4:3 aspect ratio so they can use inexpensive, mass-produced picture tubes designed for the consumer television market. Common configurations are 640×480 (VGA), 800×600 (SVGA), and 1024×768 (XGA). An XGA display with 24 bits per pixel and 25 frames/sec needs to be fed at 472 Mbps. Doubling this rate to avoid flicker is not attractive. A better solution is to transmit 25 frames/sec and have the computer store each one and paint it twice. Broadcast television does not use this strategy because television sets do not have memory, and in any event, analog signals cannot be stored in RAM without first converting them to digital form, which requires extra hardware. As a consequence, interlacing is needed for broadcast television but not for digital video.

7.3 VIDEO COMPRESSION

It should be obvious by now that manipulating multimedia material in uncompressed form is completely out of the question—it is much too big. The only hope is that massive compression is possible. Fortunately, a large body of research over the past few decades has led to many compression techniques and algorithms that make multimedia transmission feasible. In this section we will study some methods for compressing multimedia data, especially images. For more detail, see (Fluckiger, 1995; and Steinmetz and Nahrstedt, 1995).

Spotify USA Inc. Exhibit 1009 Page 175

All compression systems require two algorithms: one for compressing the data at the source, and another for decompressing it at the destination. In the literature, these algorithms are referred to as the **encoding** and **decoding** algorithms, respectively. We will use this terminology here, too.

These algorithms have certain asymmetries that are important to understand. First, for many applications, a multimedia document, say, a movie will only be encoded once (when it is stored on the multimedia server) but will be decoded thousands of times (when it is viewed by customers). This asymmetry means that it is acceptable for the encoding algorithm to be slow and require expensive hardware provided that the decoding algorithm is fast and does not require expensive hardware. On the other hand, for real-time multimedia, such as video conferencing, slow encoding is unacceptable. Encoding must happen on-the-fly, in real time.

A second asymmetry is that the encode/decode process need not be invertible. That is, when compressing a file, transmitting it, and then decompressing it, the user expects to get the original back, accurate down to the last bit. With multimedia, this requirement does not exist. It is usually acceptable to have the video signal after encoding and then decoding be slightly different than the original. When the decoded output is not exactly equal to the original input, the system is said to be **lossy**. All compression systems used for multimedia are lossy because they give much better compression.

7.3.1 The JPEG Standard

The **JPEG (Joint Photographic Experts Group)** standard for compressing continuous-tone still pictures (e.g., photographs) was developed by photographic experts working under the joint auspices of ITU, ISO, and IEC, another standards body. It is important for multimedia because, to a first approximation, the multimedia standard for moving pictures, MPEG, is just the JPEG encoding of each frame separately, plus some extra features for interframe compression and motion compensation. JPEG is defined in International Standard 10918. It has four modes and many options, but we will only be concerned with the way it is used for 24-bit RGB video and will leave out many of the details.

Step 1 of encoding an image with JPEG is block preparation. For the sake of specificity, let us assume that the JPEG input is a 640×480 RGB image with 24 bits/pixel, as shown in Fig. 7-6(a). Since using luminance and chrominance gives better compression, the luminance and two chrominance signals are computed from the RGB values. For NTSC they are called Y, I, and Q, respectively. For PAL they are called Y, U, and V, respectively, and the formulas are different. Below we will use the NTSC names, but the compression algorithm is the same.

Separate matrices are constructed for Y, I, and Q, each with elements in the range 0 to 255. Next, square blocks of four pixels are averaged in the I and Q matrices to reduce them to 320×240. This reduction is lossy, but the eye barely

Spotify USA Inc. Exhibit 1009 Page 176

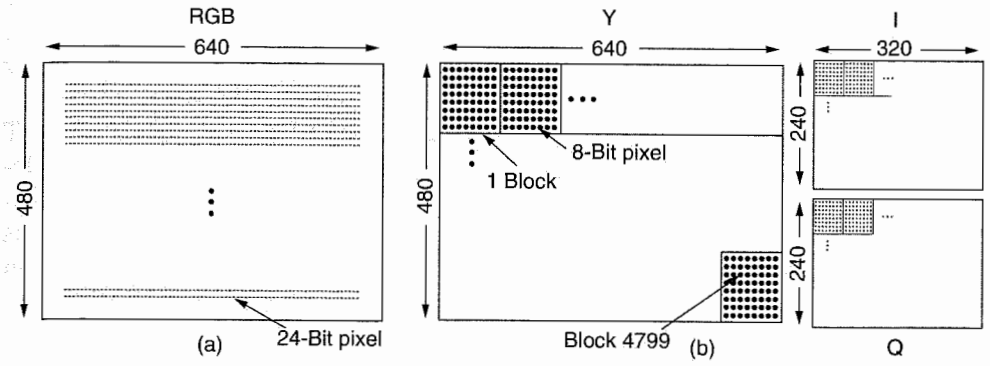


Figure 7-6. (a) RGB input data. (b) After block preparation.

notices it since the eye responds to luminance more than to chrominance. Nevertheless, it compresses the data by a factor of two. Now 128 is subtracted from each element of all three matrices to put 0 in the middle of the range. Finally, each matrix is divided up into 8×8 blocks. The Y matrix has 4800 blocks; the other two have 1200 blocks each, as shown in Fig. 7-6(b).

Step 2 of JPEG is to apply a DCT (Discrete Cosine Transformation) to each of the 7200 blocks separately. The output of each DCT is an 8×8 matrix of DCT coefficients. DCT element (0, 0) is the average value of the block. The other elements tell how much spectral power is present at each spatial frequency. In theory, a DCT is lossless, but in practice using floating-point numbers and transcendental functions always introduces some roundoff error that results in a little information loss. Normally, these elements decay rapidly with distance from the origin, (0, 0), as suggested by Fig. 7-7(b).

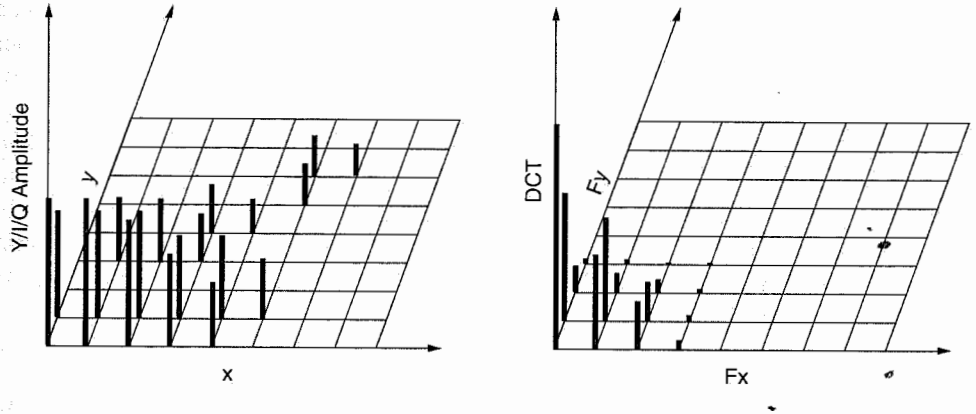


Figure 7-7. (a) One block of the Y matrix. (b) The DCT coefficients.

Once the DCT is complete, JPEG moves on to step 3, which is called **quantization**, in which the less important DCT coefficients are wiped out. This (lossy)

Spotify USA Inc. Exhibit 1009 Page 177

transformation is done by dividing each of the coefficients in the 8×8 DCT matrix by a weight taken from a table. If all the weights are 1, the transformation does nothing. However, if the weights increase sharply from the origin, higher spatial frequencies are dropped quickly.

An example of this step is given in Fig. 7-8. Here we see the initial DCT matrix, the quantization table, and the result obtained by dividing each DCT element by the corresponding quantization table element. The values in the quantization table are not part of the JPEG standard. Each application must supply its own quantization table, giving it the ability to control its own loss-compression trade-off.

DCT Coefficients

150	80	40	14	4	2	1	0
92	75	36	10	6	1	0	0
52	38	26	8	7	4	0	0
12	8	6	4	2	1	0	0
4	3	2	0	0	0	0	0
2	2	1	1	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Quantized coefficients

150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Quantization table

1	1	2	4	8	16	32	64
1	1	2	4	8	16	32	64
2	2	2	4	8	16	32	64
4	4	4	4	8	16	32	64
8	8	8	8	8	16	32	64
16	16	16	16	16	16	32	64
32	32	32	32	32	32	32	64
64	64	64	64	64	64	64	64

Figure 7-8. Computation of the quantized DCT coefficients.

Step 4 reduces the (0, 0) value of each block (the one in the upper left-hand corner) by replacing it with the amount it differs from the corresponding element in the previous block. Since these elements are the averages of their respective blocks, they should change slowly, so taking the differential values should reduce most of them to small values. No differentials are computed from the other values. The (0, 0) values are referred to as the DC components; the other values are the AC components.

Step 5 linearizes the 64 elements and applies run-length encoding to the list. Scanning the block from left to right and then top to bottom will not concentrate the zeros together, so a zig zag scanning pattern is used, as shown in Fig. 7-9. In this example, the zig zag pattern ultimately produces 38 consecutive 0s at the end of the matrix. This string can be reduced to a single count saying there are 38 zeros.

Now we have a list of numbers that represent the image (in transform space). Step 6 Huffman encodes the numbers for storage or transmission.

JPEG may seem complicated, but that is because it *is* complicated. Still, since it often produces a 20:1 compression or better, it is widely used. Decoding a JPEG image requires running the algorithm backward. JPEG is roughly symmetric: it takes about as long to decode an image as to encode it.

Spotify USA Inc. Exhibit 1009 Page 178

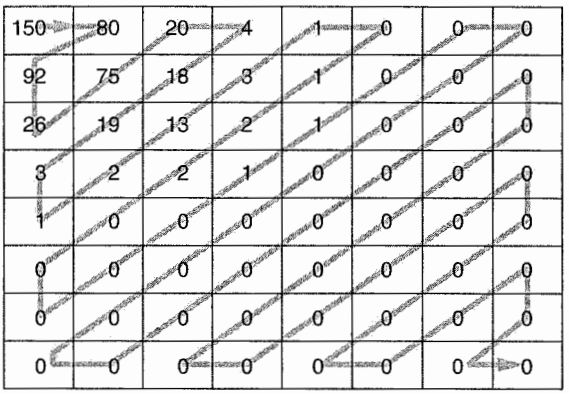


Figure 7-9. The order in which the quantized values are transmitted.

7.3.2 The MPEG Standard

Finally, we come to the heart of the matter: the **MPEG** (**Motion Picture Experts Group**) standards. These are the main algorithms used to compress videos and have been international standards since 1993. MPEG-1 (International Standard 11172) was designed for video recorder-quality output (352×240 for NTSC) using a bit rate of 1.2 Mbps. MPEG-2 (International Standard 13818) was designed for compressing broadcast quality video into 4 to 6 Mbps, so it could fit in a NTSC or PAL broadcast channel.

Both versions take advantages of the two kinds of redundancies that exist in movies: spatial and temporal. Spatial redundancy can be utilized by simply coding each frame separately with JPEG. Additional compression can be achieved by taking advantage of the fact that consecutive frames are often almost identical (temporal redundancy). The **DV** (**Digital Video**) system used by digital camcorders uses only a JPEG-like scheme because encoding has to be done in real time and it is much faster to just encode each frame separately. The consequences of this decision can be seen in Fig. 7-2: although digital camcorders have a lower data rate than uncompressed video, they are not nearly as good as full MPEG-2. (To keep the comparison honest, note that DV camcorders sample the luminance with 8 bits and each chrominance signal with 2 bits, but there is still a factor of five compression using the JPEG-like encoding.)

For scenes where the camera and background are stationary and one or two actors are moving around slowly, nearly all the pixels will be identical from frame to frame. Here, just subtracting each frame from the previous one and running JPEG on the difference would do fine. However, for scenes where the camera is panning or zooming, this technique fails badly. What is needed is some way to compensate for this motion. This is precisely what MPEG does; in fact, this is the main difference between MPEG and JPEG.

Spotify USA Inc. Exhibit 1009 Page 179

MPEG-2 output consists of three different kinds of frames that have to be processed by the viewing program:

1. I (Intracoded) frames: Self-contained JPEG-encoded still pictures.
2. P (Predictive) frames: Block-by-block difference with the last frame.
3. B (Bidirectional) frames: Differences with the last and next frame.

I-frames are just still pictures coded using JPEG, also using full-resolution luminance and half-resolution chrominance along each axis. It is necessary to have I-frames appear in the output stream periodically for three reasons. First, MPEG can be used for television broadcasting, with viewers tuning in at will. If all frames depended on their predecessors going back to the first frame, anybody who missed the first frame could never decode any subsequent frames. This would make it impossible for viewers to tune in after the movie had started. Second, if any frame were received in error, no further decoding would be possible. Third, without I-frames, while doing a fast forward or rewind, the decoder would have to calculate every frame passed over so it would know the full value of the one it stopped on. With I-frames, it is possible to skip forward or backward until an I-frame is found and start viewing there. For these reasons, I-frames are inserted into the output once or twice per second.

P-frames, in contrast, code interframe differences. They are based on the idea of **macroblocks**, which cover 16×16 pixels in luminance space and 8×8 pixels in chrominance space. A macroblock is encoded by searching the previous frame for it or something only slightly different from it.

An example of where P-frames would be useful is given in Fig. 7-10. Here we see three consecutive frames that have the same background, but differ in the position of one person. The macroblocks containing the background scene will match exactly, but the macroblocks containing the person will be offset in position by some unknown amount and will have to be tracked down.

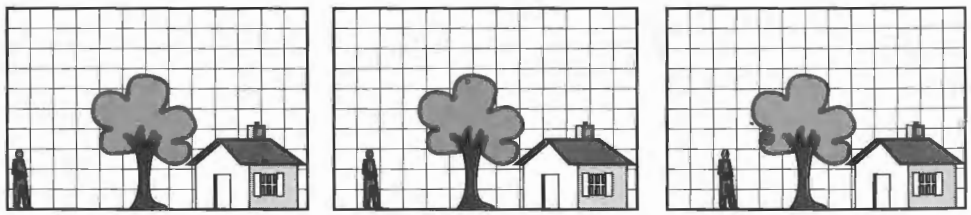


Figure 7-10. Three consecutive video frames.

The MPEG standard does not specify how to search, how far to search, or how good a match has to be to count. This is up to each implementation. For example, an implementation might search for a macroblock at the current position in the previous frame, and all other positions offset $\pm\Delta x$ in the x direction and $\pm\Delta y$

Spotify USA Inc. Exhibit 1009 Page 180

in the y direction. For each position, the number of matches in the luminance matrix could be computed. The position with the highest score would be declared the winner, provided it was above some predefined threshold. Otherwise, the macroblock would be said to be missing. Much more sophisticated algorithms are also possible, of course.

If a macroblock is found, it is encoded by taking the difference with its value in the previous frame (for luminance and both chrominances). These difference matrices are then subject to the JPEG encoding. The value for the macroblock in the output stream is then the motion vector (how far the macroblock moved from its previous position in each direction), followed by the JPEG-encoded differences with the one in the previous frame. If the macroblock is not located in the previous frame, the current value is encoded with JPEG, just as in an I-frame.

B-frames are similar to P-frames, except that they allow the reference macroblock to be in either a previous frame or in a succeeding frame, either in an I-frame or in a P-frame. This additional freedom allows improved motion compensation, and is also useful when objects pass in front of, or behind, other objects. For example, in a baseball game, when the third baseman throws the ball to first base, there may be some frame where the ball obscures the head of the moving second baseman in the background. In the next frame, the head may be partially visible to the left of the ball, with the next approximation of the head being derived from the following frame when the ball is now past the head. B-frames allow a frame to be based on a future frame.

To do B-frame encoding, the encoder needs to hold three decoded frames in memory at once: the past one, the current one, and the future one. To simplify decoding, frames must be present in the MPEG stream in dependency order, rather than in display order. Thus even with perfect timing, when a video is viewed over a network, buffering is required on the user's machine to reorder the frames for proper display. Due to this difference between dependency order and display order, trying to play a movie backward will not work without considerable buffering and complex algorithms.

7.4 MULTIMEDIA PROCESS SCHEDULING

Operating systems that support multimedia differ from traditional ones in three main ways: process scheduling, the file system, and disk scheduling. We will start with process scheduling here and continue with the other topics in subsequent sections.

7.4.1 Scheduling Homogeneous Processes

The simplest kind of video server is one that can support the display of a fixed number of movies, all using the same frame rate, video resolution, data rate, and other parameters. Under these circumstances, a simple, but effective scheduling

Spotify USA Inc. Exhibit 1009 Page 181

algorithm is as follows. For each movie, there is a single process (or thread) whose job it is to read the movie from the disk one frame at a time and then transmit that frame to the user. Since all the processes are equally important, have the same amount of work to do per frame, and block when they have finished processing the current frame, round-robin scheduling does the job just fine. The only addition needed to standard scheduling algorithms is a timing mechanism to make sure each process runs at the correct frequency.

One way to achieve the proper timing is to have a master clock that ticks at, say, 30 times per second (for NTSC). At every tick, all the processes are run sequentially, in the same order. When a process has completed its work, it issues a suspend system call that releases the CPU until the master clock ticks again. When that happens, all the processes are run again in the same order. As long as the number of processes is small enough that all the work can be done in one frame time, round-robin scheduling is sufficient.

7.4.2 General Real-Time Scheduling

Unfortunately, this model is rarely applicable in reality. The number of users changes as viewers come and go, frame sizes vary wildly due to the nature of video compression (I-frames are much larger than P- or B-frames), and different movies may have different resolutions. As a consequence, different processes may have to run at different frequencies, with different amounts of work, and with different deadlines by which the work must be completed.

These considerations lead to a different model: multiple processes competing for the CPU, each with its own work and deadlines. In the following models, we will assume that the system knows the frequency at which each process must run, how much work it has to do, and what its next deadline is. (Disk scheduling is also an issue, but we will consider that later.) The scheduling of multiple competing processes, some or all of which have deadlines that must be met is called **real-time scheduling**.

As an example of the kind of environment a real-time multimedia scheduler works in, consider the three processes, *A*, *B*, and *C* shown in Fig. 7-11. Process *A* runs every 30 msec (approximately NTSC speed). Each frame requires 10 msec of CPU time. In the absence of competition, it would run in the bursts *A*1, *A*2, *A*3, etc., each one starting 30 msec after the previous one. Each CPU burst handles one frame and has a deadline: it must complete before the next one is to start.

Also shown in Fig. 7-11 are two other processes, *B* and *C*. Process *B* runs 25 times/sec (e.g., PAL) and process *C* runs 20 times/sec (e.g., a slowed down NTSC or PAL stream intended for a user with a low-bandwidth connection to the video server). The computation time per frame is shown as 15 msec and 5 msec for *B* and *C*, respectively, just to make the scheduling problem more general than having all of them the same.

Spotify USA Inc. Exhibit 1009 Page 182

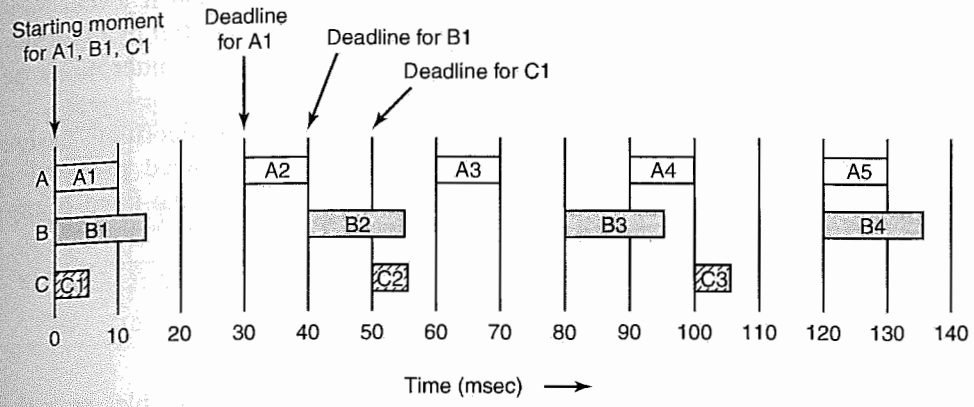


Figure 7-11. Three periodic processes, each displaying a movie. The frame rates and processing requirements per frame are different for each movie.

The scheduling question now is how to schedule A, B, and C to make sure they meet their respective deadlines. Before even looking for a scheduling algorithm, we have to see if this set of processes is schedulable at all. Recall from Sec. 2.5.4, that if process i has period P_i msec and requires C_i msec of CPU time per frame, the system is schedulable if and only if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

where m is the number of processes, in this case, 3. Note that P_i/C_i is just the fraction of the CPU being used by process i. For the example of Fig. 7-11, A is eating 10/30 of the CPU, B is eating 15/40 of the CPU, and C is eating 5/50 of the CPU. Together these fractions add to 0.808 of the CPU, so the system of processes is schedulable.

So far we assumed that there is one process per stream. Actually, there might be two (or more processes) per stream, for example, one for audio and one for video. They may run at different rates and may consume differing amounts of CPU time per burst. Adding audio processes to the mix does not change the general model, however, since all we are assuming is that there are m processes, each running at a fixed frequency with a fixed amount of work needed on each CPU burst.

In some real-time systems, processes are preemptable and in others they are not. In multimedia systems, processes are generally preemptable, meaning that a process that is in danger of missing its deadline may interrupt the running processes before the running process has finished with its frame. When it is done, the previous process can continue. This behavior is just multiprogramming, as we have seen before. We will study preemptable real-time scheduling algorithms because there is no objection to them in multimedia systems and they give better

Spotify USA Inc. Exhibit 1009 Page 183

performance than nonpreemptable ones. The only concern is that if a transmission buffer is being filled in little bursts, the buffer is completely full by the deadline so it can be sent to the user in a single operation. Otherwise jitter might be introduced.

Real-time algorithms can be either static or dynamic. Static algorithms assign each process a fixed priority in advance and then do prioritized preemptive scheduling using those priorities. Dynamic algorithms do not have fixed priorities. Below we will study an example of each type.

7.4.3 Rate Monotonic Scheduling

The classic static real-time scheduling algorithm for preemptable, periodic processes is **RMS** (**Rate Monotonic Scheduling**) (Liu and Layland, 1973). It can be used for processes that meet the following conditions:

1. Each periodic process must complete within its period.
2. No process is dependent on any other process.
3. Each process needs the same amount of CPU time on each burst.
4. Any nonperiodic processes have no deadlines.
5. Process preemption occurs instantaneously and with no overhead.

The first four conditions are reasonable. The last one is not, of course, but it makes modeling the system much easier. RMS works by assigning each process a fixed priority equal to the frequency of occurrence of its triggering event. For example, a process that must run every 30 msec (33 times/sec) gets priority 33, a process that must run every 40 msec (25 times/sec) gets priority 25, and a process that must run every 50 msec (20 times/sec) gets priority 20. The priorities are thus linear with the rate (number of times/second the process runs). This is why it is called rate monotonic. At run time, the scheduler always runs the highest priority ready process, preempting the running process if need be. Liu and Layland proved that RMS is optimal among the class of static scheduling algorithms.

Figure 7-12 shows how rate monotonic scheduling works in the example of Fig. 7-11. Processes *A*, *B*, and *C* have static priorities, 33, 25, and 20, respectively, which means that whenever *A* needs to run, it runs, preempting any other process currently using the CPU. Process *B* can preempt *C*, but not *A*. Process *C* has to wait until the CPU is otherwise idle in order to run.

In Fig. 7-12, initially all three processes are ready to run. The highest priority one, *A*, is chosen, and allowed to run until it completes at 15 msec, as shown in the RMS line. After it finishes, *B* and *C* are run in that order. Together, these processes take 30 msec to run, so when *C* finishes, it is time for *A* to run again. This rotation goes on until the system goes idle at $t = 70$.

Spotify USA Inc. Exhibit 1009 Page 184

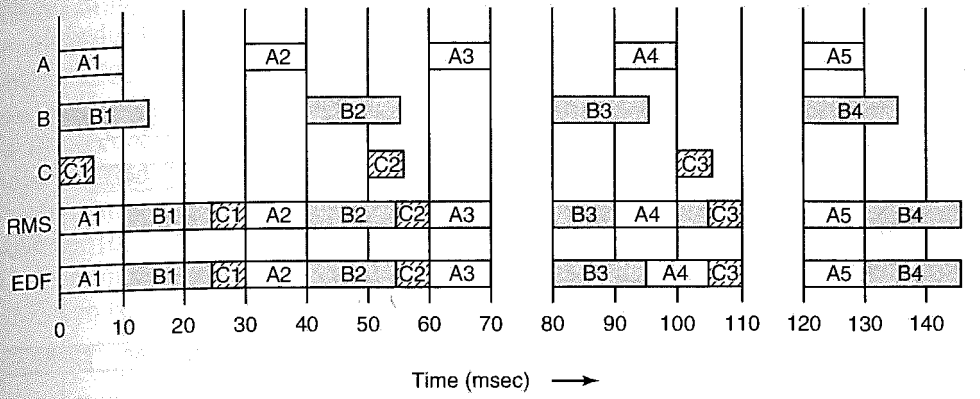


Figure 7-12. An example of RMS and EDF real-time scheduling.

At $t = 80$ B becomes ready and runs. However, at $t = 90$, a higher priority process, A, becomes ready, so it preempts B and runs until it is finished, at $t = 100$. At that point the system can choose between finishing B or starting C, so it chooses the highest priority process, B.

7.4.4 Earliest Deadline First Scheduling

Another popular real-time scheduling algorithm is **Earliest Deadline First**. EDF is a dynamic algorithm that does not require processes to be periodic, as does the rate monotonic algorithm. Nor does it require the same run time per CPU burst, as does RMS. Whenever a process needs CPU time, it announces its presence and its deadline. The scheduler keeps a list of runnable processes, sorted on deadline. The algorithm runs the first process on the list, the one with the closest deadline. Whenever a new process becomes ready, the system checks to see if its deadline occurs before that of the currently running process. If so, the new process preempts the current one.

An example of EDF is given in Fig. 7-12. Initially all three processes are ready. They are run in the order of their deadlines. A must finish by $t = 30$, B must finish by $t = 40$, and C must finish by $t = 50$, so A has the earliest deadline and thus goes first. Up until $t = 90$ the choices are the same as RMS. At $t = 90$, A becomes ready again, and its deadline is $t = 120$, the same as B's deadline. The scheduler could legitimately choose either one to run, but since in practice, preempting B has some nonzero cost associated with it, it is better to let B continue to run.

To dispel the idea that RMS and EDF always give the same results, let us now look at another example, shown in Fig. 7-13. In this example the periods of A, B, and C are the same as before, but now A needs 15 msec of CPU time per burst instead of only 10 msec. The schedulability test computes the CPU utilization as

Spotify USA Inc. Exhibit 1009 Page 185

0.500 + 0.375 + 0.100 = 0.975. Only 2.5% of the CPU is left over, but in theory the CPU is not oversubscribed and it should be possible to find a legal schedule.

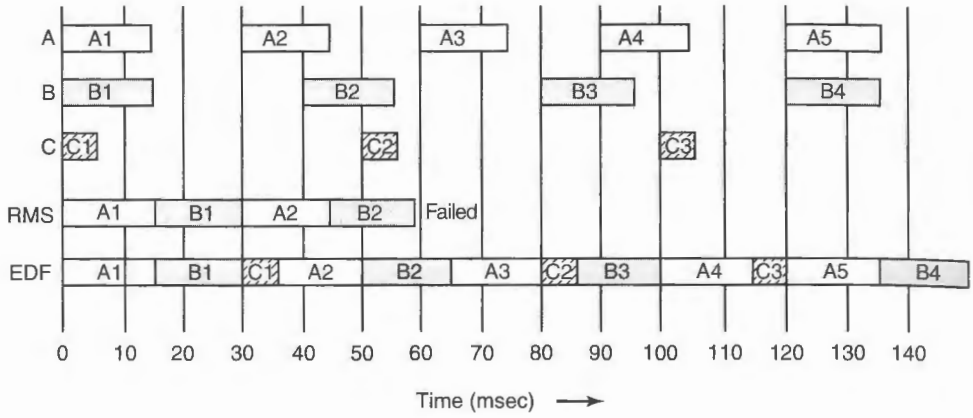


Figure 7-13. Another example of real-time scheduling with RMS and EDF.

With RMS, the priorities of the three processes are still 33, 25, and 20 as only the period matters, not the run time. This time, $B1$ does not finish until $t = 30$, at which time A is ready to roll again. By the time A is finished, at $t = 45$, B is ready again, so having a higher priority than C, it runs and C misses its deadline. RMS fails.

Now look at how EDF handles this case. At $t = 30$, there is a contest between $A2$ and $C1$. Because $C1$'s deadline is 50 and $A2$'s deadline is 60, C is scheduled. This is different from RMS, where A's higher priority wins.

At $t = 90$ A becomes ready for the fourth time. A's deadline is the same as that of the current process (120), so the scheduler has a choice of preempting or not. As before, it is better not to preempt if it is not needed, so $B3$ is allowed to complete.

In the example of Fig. 7-13, the CPU is 100% occupied up to $t = 150$. However, eventually a gap will occur because the CPU is only 97.5% utilized. Since all the starting and ending times are multiples of 5 msec, the gap will be 5 msec. In order to achieve the required 2.5% idle time, the 5 msec gap will have to occur every 200 msec, which is why it does not show up in Fig. 7-13.

An interesting question is why RMS failed. Basically, using static priorities only works if the CPU utilization is not too high. Liu and Layland (1973) proved that for any system of periodic processes, if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

then RMS is guaranteed to work. For 3, 4, 5, 10, 20, and 100, the maximum permitted utilizations are 0.780, 0.757, 0.743, 0.718, 0.705, and 0.696. As $m \rightarrow \infty$,

Spotify USA Inc. Exhibit 1009 Page 186

the maximum utilization is asymptotic to ln 2. In other words, Liu and Layland proved that for three processes, RMS always works if the CPU utilization is at or below 0.780. In our first example, it was 0.808 and RMS worked, but we were just lucky. With different periods and run times, a utilization of 0.808 might fail. In the second example, the CPU utilization was so high (0.975), there was no hope that RMS could work.

In contrast, EDF always works for any schedulable set of processes. It can achieve 100% CPU utilization. The price paid is a more complex algorithm. Thus in an actual video server, if the CPU utilization is below the RMS limit, RMS can be used. Otherwise EDF should be chosen.

7.5 MULTIMEDIA FILE SYSTEM PARADIGMS

Now that we have covered process scheduling in multimedia systems, let us continue our study by looking at multimedia file systems. These file systems use a different paradigm than traditional file systems. We will first review traditional file I/O, then turn our attention to how multimedia file servers are organized. To access a file, a process first issues an open system call. If this succeeds, the caller is given some kind of token, called a file descriptor in UNIX or a handle in Windows to be used in future calls. At that point the process can issue a read system call, providing the token, buffer address, and byte count as parameters. The operating system then returns the requested data in the buffer. Additional read calls can then be made until the process is finished, at which time it calls close to close the file and return its resources.

This model does not work well for multimedia on account of the need for real-time behavior. It works especially poorly for displaying multimedia files coming off a remote video server. One problem is that the user must make the read calls fairly precisely spaced in time. A second problem is that the video server must be able to supply the data blocks without delay, something that is difficult for it to do when the requests come in unplanned and no resources have been reserved in advance.

To solve these problems, a completely different paradigm is used by multimedia file servers: they act like VCRs (Video Cassette Recorders). To read a multimedia file, a user process issues a start system call, specifying the file to be read and various other parameters, for example, which audio and subtitle tracks to use. The video server then begins sending out frames at the required rate. It is up to the user to handle them at the rate they come in. If the user gets bored with the movie, the stop system call terminates the stream. File servers with this streaming model are often called **push servers** (because they push data at the user) and are contrasted with traditional **pull servers** where the user has to pull the data in one block at a time by repeatedly calling read to get one block after another. The difference between these two models is illustrated in Fig. 7-14.

Spotify USA Inc. Exhibit 1009 Page 187

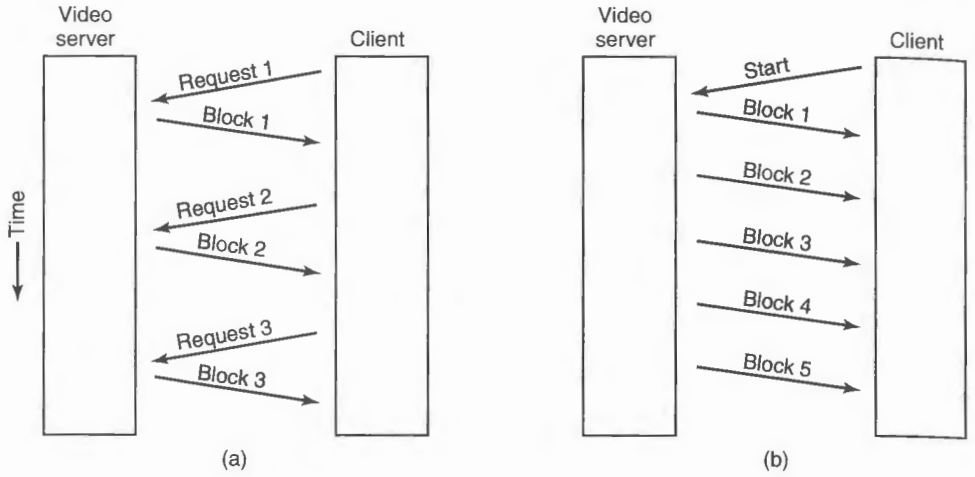


Figure 7-14. (a) A pull server. (b) A push server.

7.5.1 VCR Control Functions

Most video servers also implement standard VCR control functions, including pause, fast forward, and rewind. Pause is fairly straightforward. The user sends a message back to the video server that tells it to stop. All it has to do at that point is remember which frame goes out next. When the user tells the server to resume, it just continues from where it left off.

However, there is one complication here. To achieve acceptable performance, the server may reserve resources such as disk bandwidth and memory buffers for each outgoing stream. Continuing to tie these up while a movie is paused wastes resources, especially if the user is planning a trip to the kitchen to locate, microwave, cook, and eat a frozen pizza (especially an extra large). The resources can easily be released upon pausing, of course, but this introduces the danger that when the user tries to resume, they cannot be reacquired.

True rewind is actually easy, with no complications. All the server has to do is note that the next frame to be sent is 0. What could be easier? However, fast forward and fast backward (i.e., playing while rewinding) are much trickier. If it were not for compression, one way to go forward at 10x speed would be to just display every 10th frame. To go forward at 20x speed would require displaying every 20th frame. In fact, in the absence of compression, going forward or backward at any speed is easy. To run at k times normal speed, just display every k-th frame. To go backward at k times normal speed, do the same thing in the other direction. This approach works equally well for both pull servers and push servers.

Compression makes rapid motion either way more complicated. With a camcorder DV tape, where each frame is compressed independently of all the others,

Spotify USA Inc. Exhibit 1009 Page 188

it is possible to use this strategy, provided that the needed frame can be found quickly. Since each frame compresses by a different amount, depending on its content, each frame is a different size, so skipping ahead k frames in the file cannot be done by doing a numerical calculation. Furthermore, audio compression is done independently of video compression, so for each video frame displayed in high-speed mode, the correct audio frame must also be located (unless sound is turned off when running faster than normal). Thus fast forwarding a DV file requires an index that allows frames to be located quickly, but it is at least doable in theory.

With MPEG, this scheme does not work, even in theory, due to the use of I-, P-, and B-frames. Skipping ahead k frames (assuming that can be done at all), might land on a P-frame that is based on an I-frame that was just skipped over. Without the base frame, having the incremental changes from it (which is what a P-frame contains) is useless. MPEG requires the file to be played sequentially.

Another way to attack the problem is to actually try to play the file sequentially at 10x speed. However, doing this requires pulling data off the disk at 10x speed. At that point, the server could try to decompress the frames (something it normally does not do), figure out which frame is needed, and recompress every 10th frame as an I-frame. However, doing this puts a huge load on the server. It also requires the server to understand the compression format, something it normally does not have to know.

The alternative of actually shipping all the data over the network to the user and letting the correct frames be selected out there requires running the network at 10x speed, possibly doable, but certainly not easy given the high speed at which it normally has to operate.

All in all, there is no easy way out. The only feasible strategy requires advance planning. What can be done is build a special file containing, say, every 10th frame, and compress this file using the normal MPEG algorithm. This file is what is shown in Fig. 7-3 as "fast forward." To switch to fast forward mode, what the server must do is figure out where in the fast forward file the user currently is. For example, if the current frame is 48,210 and the fast forward file runs at 10x, the server has to locate frame 4821 in the fast forward file and start playing there at normal speed. Of course, that frame might be a P- or B-frame, but the decoding process at the client can just skip frames until it sees an I-frame. Going backward is done in an analogous way using a second specially prepared file.

When the user switches back to normal speed, the reverse trick has to be done. If the current frame in the fast forward file is 5734, the server just switches back to the regular file and continues at frame 57,340. Again, if this frame is not an I-frame, the decoding process on the client side has to ignore all frames until an I-frame is seen.

While having these two extra files does the job, the approach has some disadvantages. First, some extra disk space is required to store the additional files. Second, fast forwarding and rewinding can only be done at speeds corresponding

Spotify USA Inc. Exhibit 1009 Page 189

to the special files. Third, extra complexity is needed to switch back and forth between the regular, fast forward, and fast backward files.

7.5.2 Near Video on Demand

Having k users getting the same movie puts essentially the same load on the server as having them getting k different movies. However, with a small change in the model, great performance gains are possible. The problem with video on demand is that users can start streaming a movie at an arbitrary moment, so if there are 100 users all starting to watch some new movie at about 8 P.M., chances are that no two will start at exactly the same instant so they cannot share a stream. The change that makes optimization possible is to tell all users that movies only start on the hour and every (for example) 5 minutes thereafter. Thus if a user wants to see a movie at 8:02, he will have to wait until 8:05.

The gain here is that for a 2-hour movie, only 24 streams are needed, no matter how many customers there are. As shown in Fig. 7-15, the first stream starts at 8:00. At 8:05, when the first stream is at frame 9000, stream 2 starts. At 8:10, when the first stream is at frame 18,000 and stream 2 is at frame 9000, stream 3 starts, and so on up to stream 24, which starts at 9:55. At 10:00, stream 1 terminates and starts all over with frame 0. This scheme is called **near video on demand** because the video does not quite start on demand, but shortly thereafter.

The key parameter here is how often a stream starts. If one starts every 2 minutes, 60 streams will be needed for a two-hour movie, but the maximum waiting time to start watching will be 2 minutes. The operator has to decide how long people are willing to wait because the longer they are willing to wait, the more efficient the system, and the more movies can be shown at once. An alternative strategy is to also have a no-wait option, in which case a new stream is started on the spot, but to charge more for instant startup.

In a sense, video on demand is like using a taxi: you call it and it comes. Near video on demand is like using a bus: it has a fixed schedule and you have to wait for the next one. But mass transit only makes sense if there is a mass. In midtown Manhattan, a bus that runs every 5 minutes can count on picking up at least a few riders. A bus traveling on the back roads of Wyoming might be empty nearly all the time. Similarly, starting the latest Steven Spielberg release might attract enough customers to warrant starting a new stream every 5 minutes, but for *Gone with the Wind* it might be better to simply offer it on a demand basis.

With near video on demand, users do not have VCR controls. No user can pause a movie to make a trip to the kitchen. The best that can be done is upon returning from the kitchen, to drop back to a stream that started later, thereby repeating a few minutes of material.

Actually, there is another model for near video on demand as well. Instead of announcing in advance that some specific movie will start every 5 minutes, people can order movies whenever they want to. Every 5 minutes, the system sees which

Spotify USA Inc. Exhibit 1009 Page 190

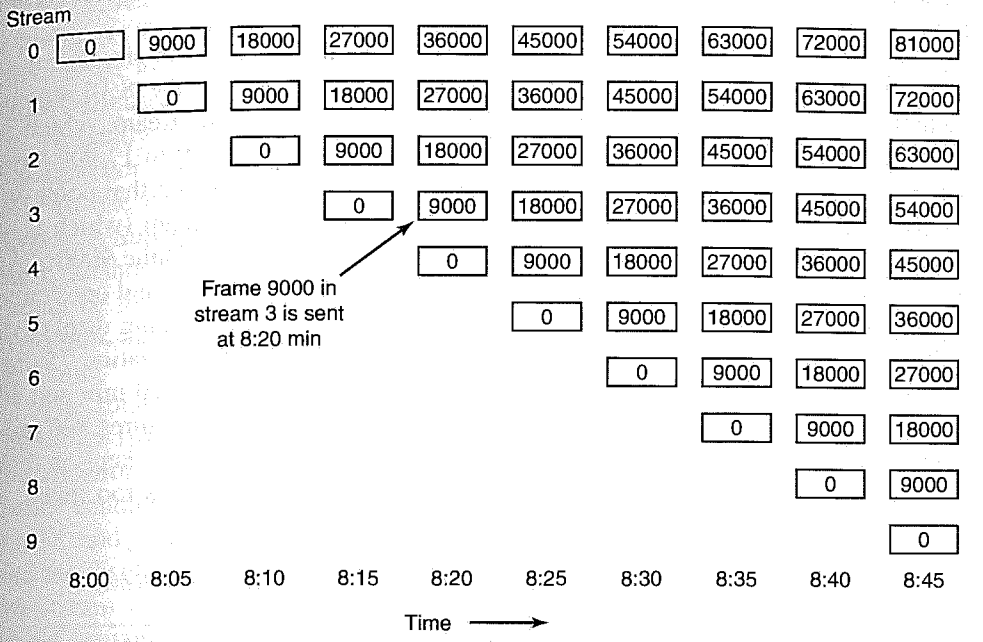


Figure 7-15. Near video on demand has a new stream starting at regular intervals, in this example every 5 minutes (9000 frames).

movies have been ordered and starts those. With this approach, a movie may start at 8:00, 8:10, 8:15, and 8:25, but not at the intermediate times, depending on demand. As a result, streams with no viewers are not transmitted, saving disk bandwidth, memory, and network capacity. On the other hand, attacking the freezer is now a bit of a gamble as there is no guarantee that there is another stream running 5 minutes behind the one the viewer was watching. Of course, the operator can provide an option for the user to display a list of all concurrent streams, but most people think their TV remote controls have more than enough buttons already and are not likely to enthusiastically welcome a few more.

7.5.3 Near Video on Demand with VCR Functions

The ideal combination would be near video on demand (for the efficiency) plus full VCR controls for every individual viewer (for the user convenience). With slight modifications to the model, such a design is possible. Below we will give a slightly simplified description of one way to achieve this goal (Abram-Profeta and Shin, 1998).

We start out with the standard near video-on-demand scheme of Fig. 7-15. However, we add the requirement that each client machine buffer the previous ΔT min and also the upcoming ΔT min locally. Buffering the previous ΔT min is

Spotify USA Inc. Exhibit 1009 Page 191

easy: just save it after displaying it. Buffering the upcoming ΔT min is harder, but can be done if clients have the ability to read two streams at once.

One way to get the buffer set up can be illustrated using an example. If a user starts viewing at 8:15, the client machine reads and displays the 8:15 stream (which is at frame 0). In parallel, it reads and stores the 8:10 stream, which is currently at the 5-min mark (i.e., frame 9000). At 8:20, frames 0 to 17,999 have been stored and the user is expecting to see frame 9000 next. From that point on, the 8:15 stream is dropped, the buffer is filled from the 8:10 stream (which is at 18,000), and the display is driven from the middle of the buffer (frame 9000). As each new frame is read, one frame is added to the end of the buffer and one frame is dropped from the beginning of the buffer. The current frame being displayed, called the **play point**, is always in the middle of the buffer. The situation 75 min into the movie is shown in Fig. 7-16(a). Here all frames between 70 min and 80 min are in the buffer. If the data rate is 4 Mbps, a 10-min buffer requires 300 million bytes of storage. With current prices, the buffer can certainly be kept on disk and possibly in RAM. If RAM is desired, but 300 million bytes is too much, a smaller buffer can be used.

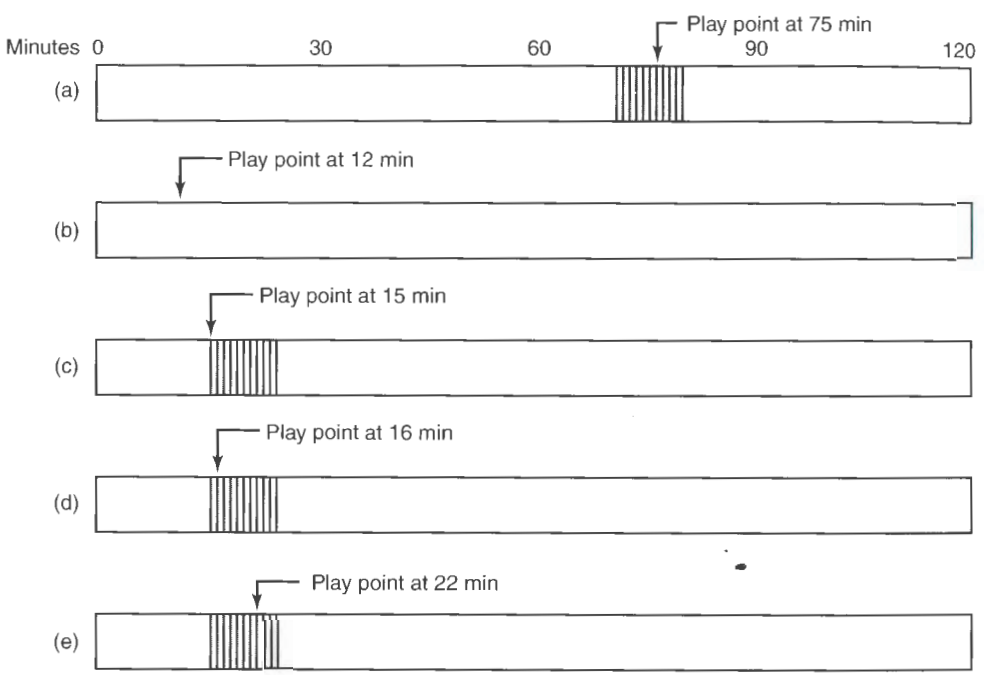


Figure 7-16. (a) Initial situation. (b) After a rewind to 12 min. (c) After waiting 3 min. (d) After starting to refill the buffer. (e) Buffer full.

Now suppose that the user decides to fast forward or fast reverse. As long as the play point stays within the range 70–80 min, the display can be fed from the

Spotify USA Inc. Exhibit 1009 Page 192

buffer. However, if the play point moves outside that interval either way, we have a problem. The solution is to turn on a private (i.e., video-on-demand) stream to service the user. Rapid motion in either direction can be handled by the techniques discussed earlier.

Normally, at some point the user will settle down and decide to watch the movie at normal speed again. At this point we can think about migrating the user over to one of the near video-on-demand streams so the private stream can be dropped. Suppose, for example, that the user decides to go back to the 12 min mark, as shown in Fig. 7-16(b). This point is far outside the buffer, so the display cannot be fed from it. Furthermore, since the switch happened (instantaneously) at 75 min, there are streams showing the movie at 5, 10, 15, and 20 min, but none at 12 min.

The solution is to continue viewing on the private stream, but to start filling the buffer from the stream currently 15 minutes into the movie. After 3 minutes, the situation is as depicted in Fig. 7-16(c). The play point is now 15 min, the buffer contains minutes 15 to 18, and the near video-on-demand streams are at 8, 13, 18, and 23 min, among others. At this point the private stream can be dropped and the display can be fed from the buffer. The buffer continues to be filled from the stream now at 18 min. After another minute, the play point is 16 min, the buffer contains minutes 15 to 19, and the stream feeding the buffer is at 19 min, as shown in Fig. 7-16(d).

After an additional 6 minutes have gone by, the buffer is full and the play point is at 22 min. The play point is not in the middle of the buffer, although that can be arranged if necessary.

7.6 FILE PLACEMENT

Multimedia files are very large, are often written only once but read many times, and tend to be accessed sequentially. Their playback must also meet strict quality of service criteria. Together, these requirements suggest different file system layouts than traditional operating systems use. We will discuss some of these issues below, first for a single disk, then for multiple disks.

7.6.1 Placing a File on a Single Disk

The most important requirement is that data can be streamed to the network or output device at the requisite speed and without jitter. For this reason, having multiple seeks during a frame is highly undesirable. One way to eliminate intrafile seeks on video servers is to use contiguous files. Normally, having files be contiguous does not work well, but on a video server that is carefully preloaded in advance with movies that do not change afterward it can work.

Spotify USA Inc. Exhibit 1009 Page 193

One complication, however, is the presence of video, audio, and text, as shown in Fig. 7-3. Even if the video, audio, and text are each stored as separate contiguous files, a seek will be needed to go from the video file to an audio file and from there to a text file, if need be. This suggests a second possible storage arrangement, with the video, audio, and text interleaved as shown in Fig. 7-17, but the entire file still contiguous. Here, the video for frame 1 is directly followed by the various audio tracks for frame 1 and then the various text tracks for frame 1. Depending on how many audio and text tracks there are, it may be simplest just to read in all the pieces for each frame in a single disk read operation and only transmit the needed parts to the user.

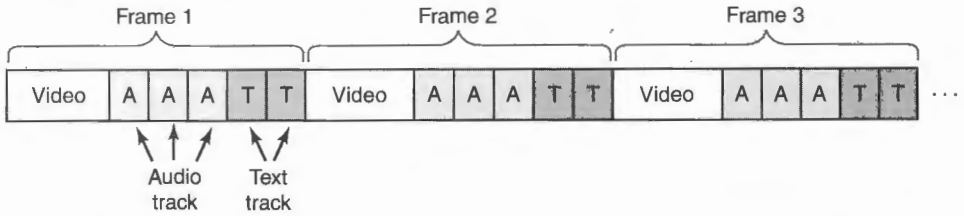


Figure 7-17. Interleaving video, audio, and text in a single contiguous file per movie.

This organization requires extra disk I/O for reading in unwanted audio and text, and extra buffer space in memory to store them. However it eliminates all seeks (on a single-user system) and does not require any overhead for keeping track of which frame is where on the disk since the whole movie is in one contiguous file. Random access is impossible with this layout, but if it is not needed, its loss is not serious. Similarly, fast forward and fast backward are impossible without additional data structures and complexity.

The advantage of having an entire movie as a single contiguous file is lost on a video server with multiple concurrent output streams because after reading a frame from one movie, the disk will have to read in frames from many other movies before coming back to the first one. Also, for a system in which movies are being written as well as being read (e.g., a system used for video production or editing), using huge contiguous files is difficult to do and not that useful.

7.6.2 Two Alternative File Organization Strategies

These observations lead to two other file placement organizations for multimedia files. The first of these, the small block model, is illustrated in Fig. 7-18(a). In this organization, the disk block size is chosen to be considerably smaller than the average frame size, even for P-frames and B-frames. For MPEG-2 at 4 Mbps with 30 frames/sec, the average frame is 16 KB, so a block size of 1 KB or 2 KB would work well. The idea here is to have a data structure, the frame index, per movie with one entry for each frame pointing to the start of

Spotify USA Inc. Exhibit 1009 Page 194

the frame. Each frame itself consists of all the video, audio, and text tracks for that frame as a contiguous run of disk blocks, as shown. In this way, reading frame k consists of indexing into the frame index to find the k-th entry, and then reading in the entire frame in one disk operation. Since different frames have different sizes, the frame size (in blocks) is needed in the frame index, but even with 1-KB disk blocks, an 8-bit field can handle a frame up to 255 KB, which is enough for an uncompressed NTSC frame, even with many audio tracks.

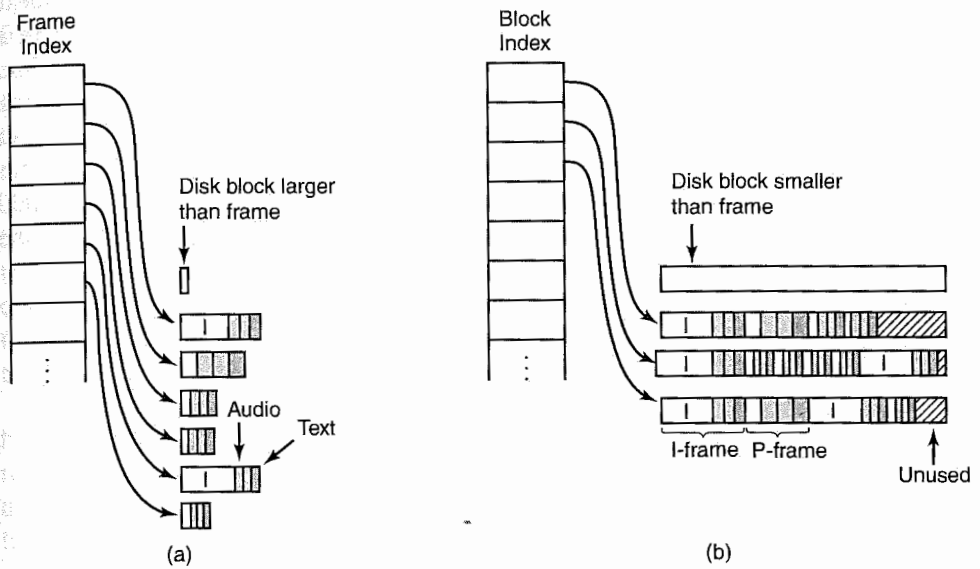


Figure 7-18. Noncontiguous movie storage. (a) Small disk blocks. (b) Large disk blocks.

The other way to store the movie is by using a large disk block (say 256 KB) and putting multiple frames in each block, as shown in Fig. 7-18(b). An index is still needed, but now it is a block index rather than a frame index. The index is, in fact, basically the same as the i-node of Fig. 6-15, possibly with the addition of information telling which frame is at the beginning of each block to make it possible to locate a given frame quickly. In general, a block will not hold an integral number of frames, so something has to be done to deal with this. Two options exist.

In the first option, which is illustrated in Fig. 7-18(b), whenever the next frame does not fit in the current block, the rest of the block is just left empty. This wasted space is internal fragmentation, the same as in virtual memory systems with fixed-size pages. On the other hand, it is never necessary to do a seek in the middle of a frame.

The other option is to fill each block to the end, splitting frames over blocks. This option introduces the need for seeks in the middle of frames, which can hurt performance, but saves disk space by eliminating internal fragmentation.

Spotify USA Inc. Exhibit 1009 Page 195

For comparison purposes, the use of small blocks in Fig. 7-18(a) also wastes some disk space because a fraction of the last block in each frame is unused. With a 1-KB disk block and a 2-hour NTSC movie consisting of 216,000 frames, the wasted disk space will only be about 108 KB out of 3.6 GB. The wasted space is harder to calculate for Fig. 7-18(b), but it will have to be much more because from time to time there will be 100 KB left at the end of a block with the next frame being an I-frame larger than that.

On the other hand, the block index is much smaller than the frame index. With a 256-KB block and an average frame of 16 KB, about 16 frames fit in a block, so a 216,000-frame movie needs only 13,500 entries in the block index, versus 216,000 for the frame index. For performance reasons, in both cases the index should list all the frames or blocks (i.e., no indirect blocks as UNIX), so tying up 13,500 8-byte entries in memory (4 bytes for the disk address, 1 byte for the frame size, and 3 bytes for the number of the starting frame) versus 216,000 5-byte entries (disk address and size only) saves almost 1 MB of RAM while the movie is playing.

These considerations lead to the following trade-offs:

1. Frame index: Heavier RAM usage while movie is playing; little disk wastage.
2. Block index (no splitting frames over blocks): Low RAM usage; major disk wastage.
3. Block index (splitting frames over blocks is allowed): Low RAM usage; no disk wastage; extra seeks

Thus the trade-offs involve RAM usage during playback, wasted disk space all the time, and performance loss during playback due to extra seeks. These problems can be attacked in various ways though. RAM usage can be reduced by paging in parts of the frame table just in time. Seeks during frame transmission can be masked by sufficient buffering, but this introduces the need for extra memory and probably extra copying. A good design has to carefully analyze all these factors and make a good choice for the application at hand.

Yet another factor here is that disk storage management is more complicated in Fig. 7-18(a) because storing a frame requires finding a consecutive run of blocks the right size. Ideally, this run of blocks should not cross a disk track boundary, but with head skew, the loss is not serious. Crossing a cylinder boundary should be avoided, however. These requirements mean that the disk's free storage has to be organized as a list of variable-sized holes, rather than a simple block list or bitmap, both of which can be used in Fig. 7-18(b).

In all cases, there is much to be said for putting all the blocks or frames of a movie within a narrow range, say a few cylinders, where possible. Such a placement means that seeks go faster so that more time will be left over for other (nonreal-time) activities or for supporting additional video streams. A constrained

Spotify USA Inc. Exhibit 1009 Page 196

placement of this sort can be achieved by dividing the disk into cylinder groups and for each group keeping separate lists or bitmaps of the free blocks. If holes are used, for example, there could be one list for 1-KB holes, one for 2-KB holes, one for holes of 3 KB to 4 KB, another for holes of size 5 KB to 8 KB, and so on. In this way it is easy to find a hole of a given size in a given cylinder group.

Another difference between these two approaches is buffering. With the small-block approach, each read gets exactly one frame. Consequently, a simple double buffering strategy works fine: one buffer for playing back the current frame and one for fetching the next one. If fixed buffers are used, each buffer has to be large enough for the biggest possible I-frame. On the other hand, if a different buffer is allocated from a pool on every frame, and the frame size is known before the frame is read in, a small buffer can be chosen for a P-frame or B-frame.

With large blocks, a more complex strategy is required because each block contains multiple frames, possibly including fragments of frames on each end of the block (depending on which option was chosen earlier). If displaying or transmitting frames requires them to be contiguous, they must be copied, but copying is an expensive operation so it should be avoided where possible. If contiguity is not required, then frames that span block boundaries can be sent out over the network or to the display device in two chunks.

Double buffering can also be used with large blocks, but using two large blocks wastes memory. One way around wasting memory is to have a circular transmission buffer slightly larger than a disk block (per stream) that feeds the network or display. When the buffer's contents drop below some threshold, a new large block is read in from the disk, the contents copied to the transmission buffer, and the large block buffer returned to a common pool. The circular buffer's size must be chosen so that when it hits the threshold, there is room for another full disk block. The disk read cannot go directly to the transmission buffer because it might have to wrap around. Here copying and memory usage are being traded off against one another.

Yet another factor in comparing these two approaches is disk performance. Using large blocks runs the disk at full speed, often a major concern. Reading in little P-frames and B-frames as separate units is not efficient. In addition, striping large blocks over multiple drives (discussed below) is possible, whereas striping individual frames over multiple drives is not.

The small-block organization of Fig. 7-18(a) is sometimes called **constant time length** because each pointer in the index represents the same number of milliseconds of playing time. In contrast, the organization of Fig. 7-18(b) is sometimes called **constant data length** because the data blocks are the same size.

Another difference between the two file organizations is that if the frame types are stored in the index of Fig. 7-18(a), it may be possible to perform a fast forward by just displaying the I-frames. However, depending on how often I-frames appear in the stream, the rate may be perceived as too fast or too slow. In any case, with the organization of Fig. 7-18(b) fast forwarding is not possible this

Spotify USA Inc. Exhibit 1009 Page 197

way. Actually reading the file sequentially to pick out the desired frames requires massive disk I/O.

A second approach is to use a special file that when played at normal speed gives the illusion of fast forwarding at 10x speed. This file can be structured the same as other files, using either a frame index or a block index. When opening a file, the system has to be able to find the fast forward file if needed. If the user hits the fast forward button, the system must instantly find and open the fast forward file and then jump to the correct place in the file. What it knows is the frame number it is currently at, but it needs the ability to locate the corresponding frame in the fast forward file. If it is currently at frame, say, 4816, and it knows the fast forward file is at 10x, then it must locate frame 482 in that file and start playing from there.

If a frame index is used, locating a specific frame is easy: just index into the frame index. If a block index is used, extra information in each entry is needed to identify which frame is in which block and a binary search of the block index has to be performed. Fast backward works in an analogous way to fast forward.

7.6.3 Placing Files for Near Video on Demand

So far we have looked at placement strategies for video on demand. For near video on demand, a different file placement strategy is more efficient. Remember that the same movie is going out as multiple staggered streams. Even if the movie is stored as a contiguous file, a seek is needed for each stream. Chen and Thapar (1997) have devised a file placement strategy to eliminate nearly all of those seeks. Its use is illustrated in Fig. 7-19 for a movie running at 30 frames/sec with a new stream starting every 5 min, as in Fig. 7-15. With these parameters, 24 concurrent streams are needed for a 2-hour movie.

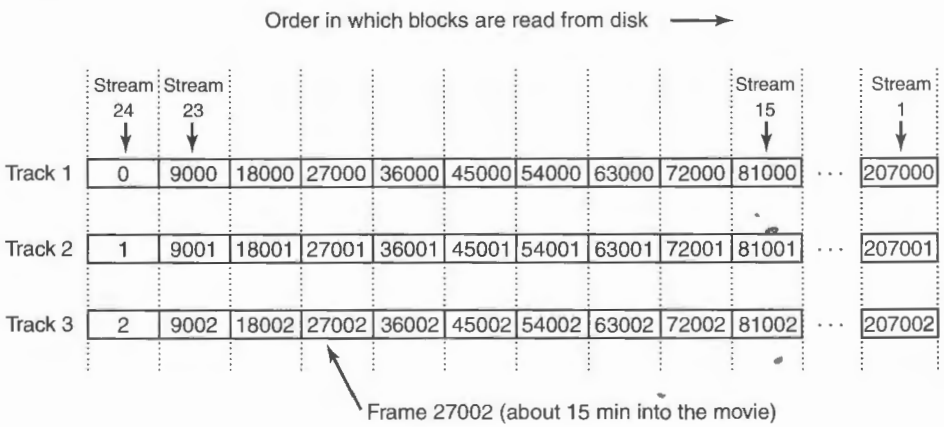


Figure 7-19. Optimal frame placement for near video on demand.

Spotify USA Inc. Exhibit 1009 Page 198

In this placement, frame sets of 24 frames are concatenated and written to the disk as a single record. They can also be read back on a single read. Consider the instant that stream 24 is just starting. It will need frame 0. Frame 23, which started 5 min earlier, will need frame 9000. Stream 22 will need frame 18,000, and so on back to stream 0 which will need frame 207,000. By putting these frames consecutively on one disk track, the video server can satisfy all 24 streams in reverse order with only one seek (to frame 0). Of course, the frames can be reversed on the disk if there is some reason to service the streams in ascending order. After the last stream has been serviced, the disk arm can move to track 2 to prepare servicing them all again. This scheme does not require the entire file to be contiguous, but still affords good performance to a number of streams at once.

A simple buffering strategy is to use double buffering. While one buffer is being played out onto 24 streams, another buffer is being loaded in advance. When the current one finishes, the two buffers are swapped and the one just used for playback is now loaded in a single disk operation.

An interesting question is how large to make the buffer. Clearly, it has to hold 24 frames. However, since frames are variable in size, it is not entirely trivial to pick the right size buffer. Making the buffer large enough for 24 I-frames is overkill, but making it large enough for 24 average frames is living dangerously.

Fortunately, for any given movie, the largest track (in the sense of Fig. 7-19) in the movie is known in advance, so a buffer of precisely that size can be chosen. However, it might just happen that in the biggest track, there are, say, 16 I-frames, whereas the next biggest track has only nine I-frames. A decision to choose a buffer large enough for the second biggest case might be wiser. Making this choice means truncating the biggest track, thus denying some streams one frame in the movie. To avoid a glitch, the previous frame can be redisplayed. No one will notice this.

Taking this approach further, if the third biggest track has only four I-frames, using a buffer capable of holding four I-frames and 20 P-frames is worth it. Introducing two repeated frames for some streams twice in the movie is probably acceptable. Where does this end? Probably with a buffer size that is big enough for 99% of the frames. Clearly, there is a trade-off here between memory used for buffers and quality of the movies shown. Note that the more simultaneous streams there are, the better the statistics are and the more uniform the frame sets will be.

7.6.4 Placing Multiple Files on a Single Disk

So far we have looked only at the placement of a single movie. On a video server, there will be many movies, of course. If they are strewn randomly around the disk, time will be wasted moving the disk head from movie to movie when multiple movies are being viewed simultaneously by different customers.

Spotify USA Inc. Exhibit 1009 Page 199

This situation can be improved by observing that some movies are more popular than others and taking popularity into account when placing movies on the disk. Although little can be said about the popularity of particular movies in general (other than noting that having big-name stars seems to help), something can be said about the relative popularity of movies in general.

For many kinds of popularity contests, such as movies being rented, books being checked out of a library, Web pages being referenced, even English words being used in a novel or the population of the largest cities, a reasonable approximation of the relative popularity follows a surprisingly predictable pattern. This pattern was discovered by a Harvard professor of linguistics, George Zipf (1902–1950) and is now called **Zipf's law**. What it states is that if the movies, books, Web pages, or words are ranked on their popularity, the probability that the next customer will choose the item ranked k-th in the list is C/k, where C is a normalization constant.

Thus the fraction of hits for the top three movies are $C/1$, $C/2$, and $C/3$, respectively, where C is computed such that the sum of all the terms is 1. In other words, if there are N movies, then

$$C/1 + C/2 + C/3 + C/4 + \cdots + C/N = 1$$

From this equation, C can be calculated. The values of C for populations with 10, 100, 1000, and 10,000 items are 0.341, 0.193, 0.134, and 0.102, respectively. For example, for 1000 movies, the probabilities for the top five movies are 0.134, 0.067, 0.045, 0.034, and 0.027, respectively.

Zipf's law is illustrated in Fig. 7-20. Just for fun, it has been applied to the populations of the 20 largest U.S. cities. Zipf's law predicts that the second largest city should have a population half of the largest city and the third largest city should be one third of the largest city, and so on. While hardly perfect, it is a surprisingly good fit.

For movies on a video server, Zipf's law states that the most popular movie is chosen twice as often as the second most popular movie, three times as often as the third most popular movie, and so on. Despite the fact that the distribution falls off fairly quickly at the beginning, it has a long tail. For example, movie 50 has a popularity of $C/50$ and movie 51 has a popularity of $C/51$, so movie 51 is 50/51 as popular as movie 50, only about a 2% difference. As one goes out further on the tail, the percent difference between consecutive movies becomes less and less. One conclusion is that the server needs a lot of movies since there is substantial demand for movies outside the top 10.

Knowing the relative popularities of the different movies makes it possible to model the performance of a video server and to use that information for placing files. Studies have shown that the best strategy is surprisingly simple and distribution independent. It is called the **organ-pipe algorithm** (Grossman and Silverman, 1973; and Wong, 1983). It consists of placing the most popular movie in the middle of the disk, with the second and third most popular movies on either side

Spotify USA Inc. Exhibit 1009 Page 200

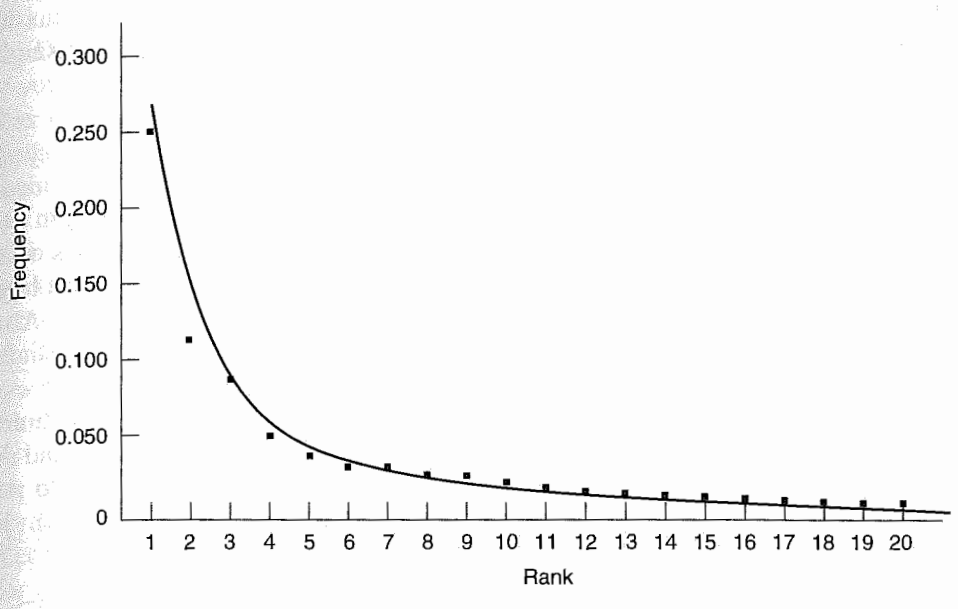


Figure 7-20. The curve gives Zipf's law for $N = 20$. The squares represent the populations of the 20 largest cities in the U.S., sorted on rank order (New York is 1, Los Angeles is 2, Chicago is 3, etc.).

of it. Outside of these come numbers four and five, and so on, as shown in Fig. 7-21. This placement works best if each movie is a contiguous file of the type shown in Fig. 7-17, but can also be used to some extent if each movie is constrained to a narrow range of cylinders. The name of the algorithm comes from the fact that a histogram of the probabilities looks like a slightly lopsided organ.

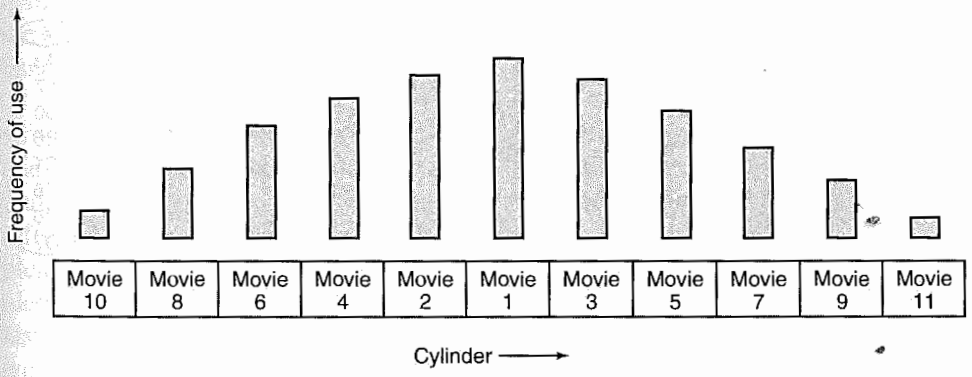


Figure 7-21. The organ-pipe distribution of files on a video server

What this algorithm does is try to keep the disk head in the middle of the disk. With 1000 movies and a Zipf's law distribution, the top five movies represent a

Spotify USA Inc. Exhibit 1009 Page 201

total probability of 0.307, which means that the disk head will stay in the cylinders allocated to the top five movies about 30% of the time, a surprisingly large amount if 1000 movies are available.

7.6.5 Placing Files on Multiple Disks

To get higher performance, video servers often have many disks that can run in parallel. Sometimes RAIDs are used, but often not because what RAIDs offer is higher reliability at the cost of performance. Video servers generally want high performance and do not care so much about correcting transient errors. Also RAID controllers can become a bottleneck if they have too many disks to handle at once.

A more common configuration is simply a large number of disks, sometimes referred to as a **disk farm**. The disks do not rotate in a synchronized way and do not contain any parity bits, as RAIDS do. One possible configuration is to put movie *A* on disk 1, movie *B* on disk 2, and so on, as shown in Fig. 7-22(a). In practice, with modern disks several movies can be placed on each disk.

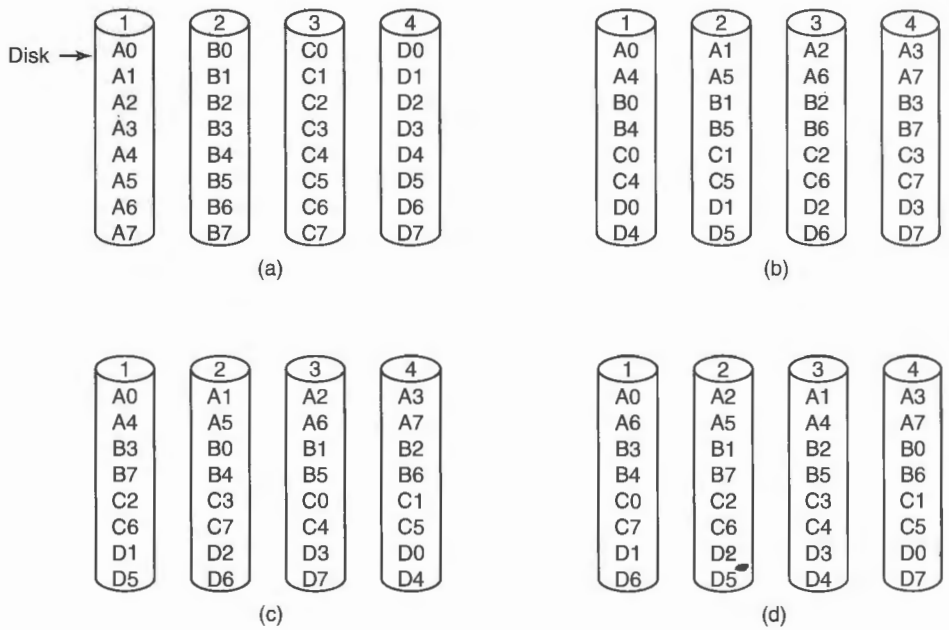


Figure 7-22. Four ways of organizing multimedia files over multiple disks. (a) No striping. (b) Same striping pattern for all files. (c) Staggered striping. (d) Random striping.

This organization is simple to implement and has straightforward failure characteristics: if one disk fails, all the movies on it become unavailable. Note that a company losing a disk full of movies is not nearly as bad as a company losing a

Spotify USA Inc. Exhibit 1009 Page 202

disk full of data because the movies can easily be reloaded on a spare disk from a DVD. A disadvantage of this approach is that the load may not be well balanced. If some disks hold movies that are currently much in demand and other disks hold less popular movies, the system will not be fully utilized. Of course, once the usage frequencies of the movies are known, it may be possible to move some of them to balance the load by hand.

A second possible organization is to stripe each movie over multiple disks, four in the example of Fig. 7-22(b). Let us assume for the moment that all frames are the same size (i.e., uncompressed). A fixed number of bytes from movie *A* is written to disk 1, then the same number of bytes is written to disk 2, and so on until the last disk is reached (in this case with unit *A*3). Then the striping continues at the first disk again with *A*4 and so on until the entire file has been written. At that point movies *B*, *C*, and *D* are striped using the same pattern.

A possible disadvantage of this striping pattern is that because all movies start on the first disk, the load across the disks may not be balanced. One way to spread the load better is to stagger the starting disks, as shown in Fig. 7-22(c). Yet another way to attempt to balance the load is to use a random striping pattern for each file, as shown in Fig. 7-22(d).

So far we have assumed that all frames are the same size. With MPEG-2 movies, this assumption is false: I-frames are much larger than P-frames. There are two ways of dealing with this complication: stripe by frame or stripe by block. When striping by frame, the first frame of movie *A* goes on disk 1 as a contiguous unit, independent of how big it is. The next frame goes on disk 2, and so on. Movie *B* is striped in a similar way, either starting at the same disk, the next disk (if staggered), or a random disk. Since frames are read one at a time, this form of striping does not speed up the reading of any given movie. However, it spreads the load over the disks much better than in Fig. 7-22(a), which may behave badly if many people decide to watch movie *A* tonight and nobody wants movie *C*. On the whole, spreading the load over all the disks makes better use of the total disk bandwidth, and thus increases the number of customers that can be served.

The other way of striping is by block. For each movie, fixed-size units are written on each of the disks in succession (or at random). Each block contains one or more frames or fragments thereof. The system can now issue requests for multiple blocks at once for the same movie. Each request asks to read data into a different memory buffer, but in such a way that when all requests have been completed, a contiguous chunk of the movie (containing many frames) is now assembled in memory contiguously. These requests can proceed in parallel. When the last request has been satisfied, the requesting process can be signaled that the work has been completed. It can then begin transmitting the data to the user. A number of frames later, when the buffer is down to the last few frames, more requests are issued to preload another buffer. This approach uses large amounts of memory for buffering in order to keep the disks busy. On a system with 1000 active users and 1-MB buffers (for example, using 256-KB blocks on each of four

Spotify USA Inc. Exhibit 1009 Page 203

disks), 1 GB of RAM is needed for the buffers. Such an amount is small potatoes on a 1000-user server and should not be a problem.

One final issue concerning striping is how many disks to stripe over. At one extreme, each movie is striped over all the disks. For example, with 2-GB movies and 1000 disks, a block of 2 MB could be written on each disk so that no movie uses the same disk twice. At the other extreme, the disks are partitioned into small groups (as in Fig. 7-22) and each movie is restricted to a single partition. The former, called **wide striping**, does a good job of balancing the load over the disks. Its main problem is that if every movie uses every disk and one disk goes down, no movie can be shown. The latter, called **narrow striping**, may suffer from hot spots (popular partitions), but loss of one disk only ruins the movies in its partition. Striping of variable-sized frames is analyzed in detail mathematically in (Shenoy and Vin, 1999).

7.7 CACHING

Traditional LRU file caching does not work well with multimedia files because the access patterns for movies are different from those of text files. The idea behind traditional LRU buffer caches is that after a block is used, it should be kept in the cache in case it is needed again quickly. For example, when editing a file, the set of blocks on which the file is written tend to be used over and over until the edit session is finished. In other words, when there is relatively high probability that a block will be reused within a short interval, it is worth keeping around to eliminate a future disk access.

With multimedia, the usual access pattern is that a movie is viewed from beginning to end sequentially. A block is unlikely to be used a second time unless the user rewinds the movie to see some scene again. Consequently, normal caching techniques do not work. However, caching can still help, but only if used differently. In the following sections we will look at caching for multimedia.

7.7.1 Block Caching

Although just keeping a block around in the hope that it may be reused quickly is pointless, the predictability of multimedia systems can be exploited to make caching useful again. Suppose that two users are watching the same movie, with one of them having started 2 sec after the other. After the first user has fetched and viewed any given block, it is very likely that the second user will need the same block 2 sec later. The system can easily keep track of which movies have only one viewer and which have two or more viewers spaced closely together in time.

Thus whenever a block is read on behalf of a movie that will be needed again shortly, it may make sense to cache it, depending on how long it has to be cached

Spotify USA Inc. Exhibit 1009 Page 204

and how tight memory is. Instead of keeping all disk blocks in the cache and discarding the least recently used one when the cache fills up, a different strategy should be used. Every movie that has a second viewer within some time ΔT of the first viewer can be marked as cachable and all its blocks cached until the second (and possibly third) viewer has used them. For other movies, no caching is done at all.

This idea can be taken a step further. In some cases it may be feasible to merge two streams. Suppose that two users are watching the same movie but with a 10-sec delay between them. Holding the blocks in the cache for 10 sec is possible but wastes memory. An alternative, but slightly sneaky, approach is to try to get the two movies in sync. This can be done by changing the frame rate for both movies. This idea is illustrated in Fig. 7-23.

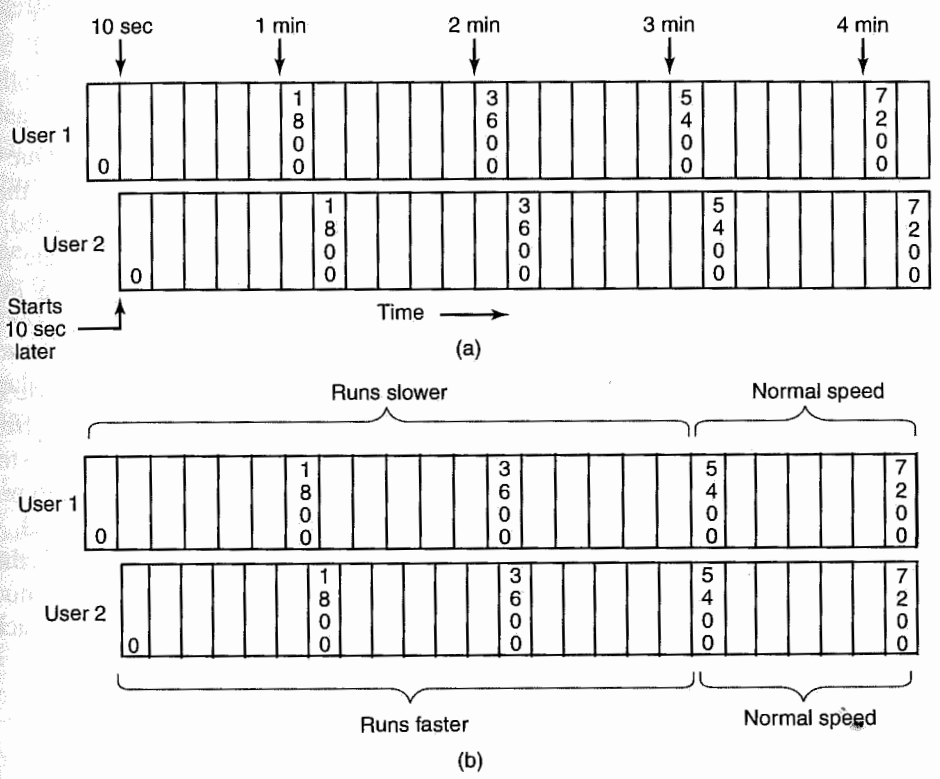


Figure 7-23. (a) Two users watching the same movie 10 sec out of sync. (b) Merging the two streams into one.

In Fig. 7-23(a), both movies run at the standard NTSC rate of 1800 frames/min. Since user 2 started 10 sec later, he continues to be 10 sec beyond for the entire movie. In Fig. 7-23(b), however, user 1's stream is slowed down when user 2 shows up. Instead of running 1800 frames/min, for the next 3 min, it

Spotify USA Inc. Exhibit 1009 Page 205

runs at 1750 frames/min. After 3 minutes, it is at frame 5550. In addition, user 2's stream is played at 1850 frames/min for the first 3 min, also putting it at frame 5550. From that point on, both play at normal speed.

During the catch-up period, user 1's stream is running 2.8% slow and user 2's stream is running 2.8% fast. It is unlikely that the users will notice this. However, if that is a concern, the catch-up period can be spread out over a longer interval than 3 minutes.

An alternative way to slow down a user to merge with another stream is to give users the option of having commercials in their movies, presumably for a lower viewing price than commercial-free movies. The user can also choose the product categories, so the commercials will be less intrusive and more likely to be watched. By manipulating the number, length, and timing of the commercials, the stream can be held back long enough to get in sync with the desired stream (Krishnan, 1999).

7.7.2 File Caching

Caching can also be useful in multimedia systems in a different way. Due to the large size of most movies (2 GB), video servers often cannot store all their movies on disk, so they keep them on DVD or tape. When a movie is needed, it can always be copied to disk, but there is a substantial startup time to locate the movie and copy it to disk. Consequently, most video servers maintain a disk cache of the most heavily requested movies. The popular movies are stored in their entirety on disk.

Another way to use caching is to keep the first few minutes of each movie on disk. That way, when a movie is requested, playback can start immediately from the disk file. Meanwhile, the movie is copied from DVD or tape to disk. By storing enough of the movie on disk all the time, it is possible to have a very high probability that the next piece of the movie has been fetched before it is needed. If all goes well, the entire movie will be on disk well before it is needed. It will then go in the cache and stay on disk in case there are more requests later. If too much time goes by without another request, the movie will be removed from the cache to make room for a more popular one.

7.8 DISK SCHEDULING FOR MULTIMEDIA

Multimedia puts different demands on the disks than traditional text-oriented applications such as compilers or word processors. In particular, multimedia demands an extremely high data rate and real-time delivery of the data. Neither of these is trivial to provide. Furthermore, in the case of a video server, there is economic pressure to have a single server handle thousands of clients simultaneously. These requirements impact the entire system. Above we looked at the file system. Now let us look at disk scheduling for multimedia.

Spotify USA Inc. Exhibit 1009 Page 206

7.8.1 Static Disk Scheduling

Although multimedia puts enormous real-time and data-rate demands on all parts of the system, it also has one property that makes it easier to handle than a traditional system: predictability. In a traditional operating system, requests are made for disk blocks in a fairly unpredictable way. The best the disk subsystem can do is perform a one-block read ahead for each open file. Other than that, all it can do is wait for requests to come in and process them on demand. Multimedia is different. Each active stream puts a well-defined load on the system that is highly predictable. For NTSC playback, every 33.3 msec, each client wants the next frame in its file and the system has 33.3 msec to provide all the frames (the system needs to buffer at least one frame per stream so that the fetching of frame $k + 1$ can proceed in parallel with the playback of frame k).

This predictable load can be used to schedule the disk using algorithms tailored to multimedia operation. Below we will consider just one disk, but the idea can be applied to multiple disks as well. For this example we will assume that there are 10 users, each one viewing a different movie. Furthermore, we will assume that all movies have the same resolution, frame rate, and other properties.

Depending on the rest of the system, the computer may have 10 processes, one per video stream, or one process with 10 threads, or even one process with one thread that handles the 10 streams in round-robin fashion. The details are not important. What is important, is that time is divided up into **rounds**, where a round is the frame time (33.3 msec for NTSC, 40 msec for PAL). At the start of each round, one disk request is generated on behalf of each user, as shown in Fig. 7-24.

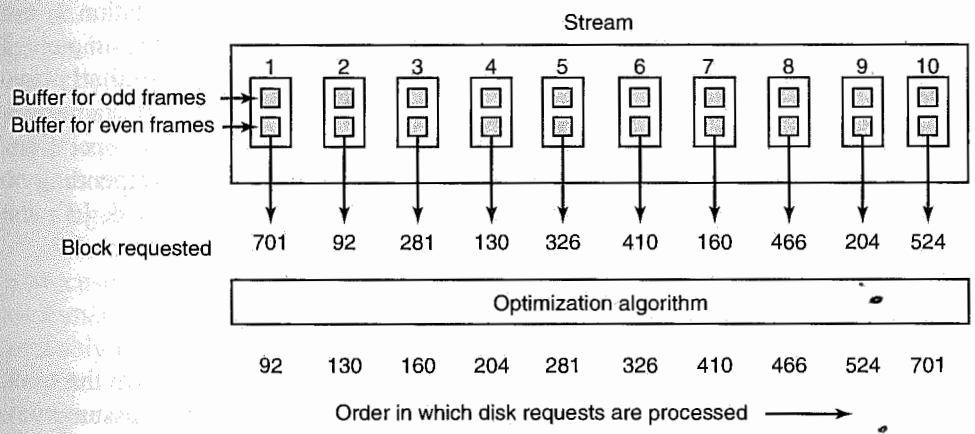


Figure 7-24. In one round, each movie asks for one frame.

After all the requests have come in at the start of the round, the disk knows what it has to do during that round. It also knows that no other requests will come in until these have been processed and the next round has begun. Consequently, it

can sort the requests in the optimal way, probably in cylinder order (although conceivably in sector order in some cases) and then process them in the optimal order. In Fig. 7-24, the requests are shown sorted in cylinder order.

At first glance, one might think that optimizing the disk in this way has no value because as long as the disk meets the deadline, it does not matter if it meets it with 1 msec to spare or 10 msec to spare. However, this conclusion is false. By optimizing seeks in this fashion, the average time to process each request is diminished, which means that the disk can handle more streams per round on the average. In other words, optimizing disk requests like this increases the number of movies the server can transmit simultaneously. Spare time at the end of the round can also be used to service any nonreal-time requests that may exist.

If a server has too many streams, once in a while when it is asked to fetch frames from distant parts of the disk and miss a deadline. But as long as missed deadlines are rare enough, they can be tolerated in return for handling more streams at once. Note that what matters is the number of streams being fetched. Having two or more clients per stream does not affect disk performance or scheduling.

To keep the flow of data out to the clients moving smoothly, double buffering is needed in the server. During round 1, one set of buffers is used, one buffer per stream. When the round is finished, the output process or processes are unblocked and told to transmit frame 1. At the same time, new requests come in for frame 2 of each movie (there might be a disk thread and an output thread for each movie). These requests must be satisfied using a second set of buffers, as the first ones are still busy. When round 3 starts, the first set of buffers are now free and can be reused to fetch frame 3.

We have assumed that there is one round per frame. This limitation is not strictly necessary. There could be two rounds per frame to reduce the amount of buffer space required, at the cost of twice as many disk operations. Similarly, two frames could be fetched from the disk per round (assuming pairs of frames are stored contiguously on the disk). This design cuts the number of disk operations in half, at the cost of doubling the amount of buffer space required. Depending on the relative availability, performance, and cost of memory versus disk I/O, the optimum strategy can be calculated and used.

7.8.2 Dynamic Disk Scheduling

In the example above, we made the assumption that all streams have the same resolution, frame rate, and other properties. Now let us drop this assumption. Different movies may now have different data rates, so it is not possible to have one round every 33.3 msec and fetch one frame for each stream. Requests come in to the disk more or less at random.

Each read request specifies which block is to be read and in addition at what time the block is needed, that is, the deadline. For simplicity, we will assume that

Spotify USA Inc. Exhibit 1009 Page 208

the actual service time for each request is the same (even though this is certainly not true). In this way we can subtract the fixed service time from each request to get the latest time the request can be initiated and still meet the deadline. This makes the model simpler because what the disk scheduler cares about is the deadline for scheduling the request.

When the system starts up, there are no disk requests pending. When the first request comes in, it is serviced immediately. While the first seek is taking place, other requests may come in, so when the first request is finished, the disk driver may have a choice of which request to process next. Some request is chosen and started. When that request is finished, there is again a set of possible requests: those that were not chosen the first time and the new arrivals that came in while the second request was being processed. In general, whenever a disk request completes, the driver has some set of requests pending from which it has to make a choice. The question is: "What algorithm does it use to select the next request to service?"

Two factors play a role in selecting the next disk request: deadlines and cylinders. From a performance point of view, keeping the requests sorted on cylinder and using the elevator algorithm minimizes total seek time, but may cause requests on outlying cylinders to miss their deadline. From a real-time point of view, sorting the requests on deadline and processing them in deadline order, earliest deadline first, minimizes the chance of missing deadlines, but increases total seek time.

These factors can be combined using the **scan-EDF algorithm** (Reddy and Wyllie, 1992). The basic idea of this algorithm is to collect requests whose deadlines are relatively close together into batches and process these in cylinder order. As an example, consider the situation of Fig. 7-25 at $t = 700$. The disk driver knows it has 11 requests pending for various deadlines and various cylinders. It could decide, for example, to treat the five requests with the earliest deadlines as a batch, sort them on cylinder number, and use the elevator algorithm to service these in cylinder order. The order would then be 110, 330, 440, 676, and 680. As long as every request is completed before its deadline, the requests can be safely rearranged to minimize the total seek time required.

When different streams have different data rates, a serious issue arises when a new customer shows up: should the customer be admitted. If admission of the customer will cause other streams to miss their deadlines frequently, the answer is probably no. There are two ways to calculate whether to admit the new customer or not. One way is to assume that each customer needs a certain amount of resources on the average, for example, disk bandwidth, memory buffers, CPU time, etc. If there is enough of each left for an average customer, the new one is admitted.

The other algorithm is more detailed. It takes a look at the specific movie the new customer wants and looks up the (precomputed) data rate for that movie, which differs for black and white versus color, cartoons versus filmed, and even

Spotify USA Inc. Exhibit 1009 Page 209

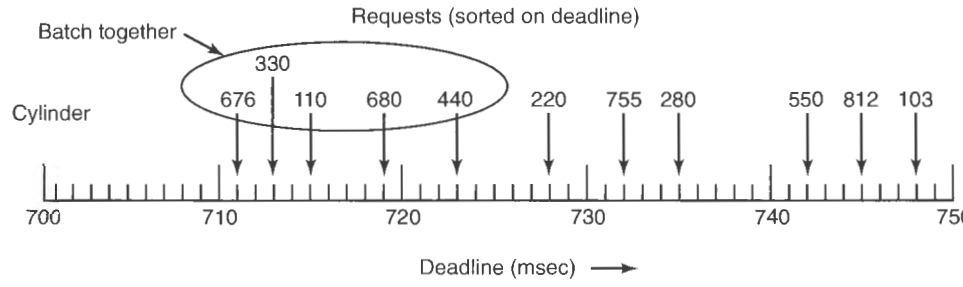


Figure 7-25. The scan-EDF algorithm uses deadlines and cylinder numbers for scheduling.

love stories versus war films. Love stories move slowly with long scenes and slow cross dissolves, all of which compress well whereas war films have many rapid cuts, and fast action, hence many I-frames and large P-frames. If the server has enough capacity for the specific film the new customer wants, then admission is granted; otherwise it is denied.

7.9 RESEARCH ON MULTIMEDIA

Multimedia is a hot topic these days, so there is a considerable amount of research about it. Much of this research is about the content, construction tools, and applications, all of which are beyond the scope of this book. However, some of it involves operating system structure, either writing a new multimedia operating system (Brandwein et al., 1994), or adding multimedia support to an existing operating system (Mercer, 1994). A related area is the design of multimedia servers (Bernhardt and Biersack, 1996; Heybey et al., 1996; Lougher et al., 1994; and Wong and Lee, 1997).

Some papers on multimedia are not about complete new systems, but about algorithms useful in multimedia systems. A popular topic has been real-time CPU scheduling for multimedia (Baker-Harvey, 1999; Bolosky et al., 1997; Dan et al., 1994; Goyal et al., 1996; Jones et al., 1997; Nieh and Lam, 1997; and Wu and Shu, 1996). Another topic that has been examined is disk scheduling for multimedia (Lee et al., 1997; Rompogiannakis et al., 1998; and Wang et al., 1999). File placement and load management on video servers are also important (Gafsi and Biersack, 1999; Shenoy and Vin, 1999; Shenoy et al., 1999; and Venkatasubramanian and Ramanathan, 1997) as is merging video streams to reduce bandwidth requirements (Eager et al., 1999).

In the text, we discussed how movie popularity affects placement on the video server. This topic is an ongoing area of research (Bisdikian and Patel, 1995; and

Spotify USA Inc. Exhibit 1009 Page 210

Griwodz et al., 1997). Finally, security and privacy in multimedia (e.g., in videoconferencing) are also subjects of research interest (Adams and Sasse, 1999; and Honeyman et. al, 1998)

7.10 SUMMARY

Multimedia is an up-and-coming use of computers. Due to the large sizes of multimedia files and their stringent real-time playback requirements, operating systems designed for text are not optimal for multimedia. Multimedia files consist of multiple, parallel tracks, usually one video and at least one audio and sometimes subtitle tracks as well. These must all be synchronized during playback.

Audio is recorded by sampling the volume periodically, usually 44,100 times/sec (for CD quality sound). Compression can be applied to the audio signal, giving a uniform compression rate of about 10x. Video compression uses both intraframe compression (JPEG) and interframe compression (MPEG). The latter represents P-frames as differences from the previous frame. B-frames can be based either on the previous frame or the next frame.

Multimedia needs real-time scheduling in order to meet its deadlines. Two algorithms are commonly used. The first is rate monotonic scheduling, which is a static preemptive algorithm that assigns fixed priorities to processes based on their periods. The second is earliest deadline first, which is a dynamic algorithm that always chooses the process with the closest deadline. EDF is more complicated, but it can achieve 100% utilization, something that RMS cannot achieve.

Multimedia file systems usually use a push model rather than a pull model. Once a stream is started, the bits come off the disk without further user requests. This approach is radically different from conventional operating systems, but is needed to meet the real-time requirements.

Files can be stored contiguously or not. In the latter case, the unit can be variable length (one block is one frame) or fixed length (one block is many frames). These approaches have different trade-offs.

File placement on the disk affects performance. When there are multiple files, the organ-pipe algorithm is sometimes used. Striping files across multiple disks, either wide or narrow, is common. Block and file caching strategies are also widely employed to improve performance.

PROBLEMS

1. What is the bit rate for uncompressed full-color XGA running at 25 frames/sec? Can a stream at this rate come off an UltraWide SCSI disk?

Spotify USA Inc. Exhibit 1009 Page 211

2. Can uncompressed black-and-white NTSC television be sent over fast Ethernet? If so, how many channels at once?

3. HDTV has twice the horizontal resolution of regular TV (1280 versus 640 pixels). Using information provided in the text, how much more bandwidth does it require than standard TV?

4. In Fig. 7-3, there are separate files for fast forward and fast reverse. If a video server is intended to support slow motion as well, is another file required for slow motion in the forward direction? What about in the backward direction?

5. A Compact Disc holds 74 min of music or 650 MB of data. Make an estimate of the compression factor used for music.

6. A sound signal is sampled using a signed 16-bit number (1 sign bit, 15 magnitude bits). What is the maximum quantization noise in percent? Is this a bigger problem for flute concertos or for rock and roll or is it the same for both? Explain your answer.

7. A recording studio is able to make a master digital recording using 20-bit sampling. The final distribution to listeners will use 16 bits. Suggest a way to reduce the effect of quantization noise, and discuss advantages and disadvantages of your scheme.

8. NTSC and PAL both use a 6-MHz broadcast channel, yet NTSC has 30 frames/sec whereas PAL has only 25 frames/sec. How is this possible? Does this mean that if both systems were to use the same color encoding scheme, NTSC would have inherently better quality than PAL? Explain your answer.

9. The DCT transformation uses an 8×8 block, yet the algorithm used for motion compensation uses 16×16. Does this difference cause problems, and if so, how are they solved in MPEG?

10. In Fig. 7-10 we saw how MPEG works with a stationary background and a moving actor. Suppose that an MPEG video is made from a scene in which the camera is mounted on a tripod and pans slowing from left to right at a speed such that no two consecutive frames are the same. Do all the frames have to be I-frames now? Why or why not?

11. Suppose that each of the three processes in Fig. 7-11 is accompanied by a process that supports an audio stream running with the same period as its video process, so audio buffers can be updated between video frames. All three of these audio processes are identical. How much CPU time is available for each burst of an audio process?

12. Two real-time processes are running on a computer. The first one runs every 25 msec for 10 msec. The second one runs every 40 msec for 15 msec. Will RMS always work for them?

13. The CPU of a video server has a utilization of 65%. How many movies can it show using RMS scheduling?

14. In Fig. 7-13, EDF keeps the CPU busy 100% of the time up to $t = 150$. It cannot keep the CPU busy indefinitely because there is only 975-msec work per second for it to do so. Extend the figure beyond 150 msec and determine when the CPU first goes idle with EDF.

Spotify USA Inc. Exhibit 1009 Page 212

15. A DVD can hold enough data for a full-length movie and the transfer rate is adequate to display a television-quality program. Why not just use a "farm" of many DVD drives as the data source for a video server?

16. The operators of a near video-on-demand system have discovered that people in a certain city are not willing to wait more than 6 minutes for a movie to start. How many parallel streams do they need for a 3-hour movie?

17. Consider a system using the scheme of Abram-Profeta and Shin in which the video server operator wishes customers to be able to search forward or backward for 1 min entirely locally. Assuming the video stream is MPEG-2 at 4 Mbps, how much buffer space must each customer have locally?

18. A video-on-demand system for HDTV uses the small block model of Fig. 7-18(a) with a 1-KB disk block. If the video resolution is 1280×720 and the data stream is 12 Mbps, how much disk space is wasted on internal fragmentation in a 2-hour movie using NTSC?

19. Consider the storage allocation scheme of Fig. 7-18(a) for NTSC and PAL. For a given disk block and movie size, does one of them suffer more internal fragmentation than the other? If so, which one is better and why?

20. Consider the two alternatives shown in Fig. 7-18. Does the shift toward HDTV favor either of these systems over the other? Discuss.

21. The near video-on-demand scheme of Chen and Thapar works best when each frame set is the same size. Suppose that a movie is being shown in 24 simultaneous streams and that one frame in 10 is an I-frame. Also assume that I-frames are 10 times larger than P-frames. B-frames are the same size as P-frames. What is the probability that a buffer equal to 4 I-frames and 20 P-frames will not be big enough? Do you think that such a buffer size is acceptable? To make the problem tractable, assume that frame types are randomly and independently distributed over the streams.

22. The end result of Fig. 7-16 is that the play point is not in the middle of the buffer any more. Devise a scheme to have at least 5 min behind the play point and 5 min ahead of it. Make any reasonable assumptions you have to, but state them explicitly.

23. The design of Fig. 7-17 requires that all language tracks be read on each frame. Suppose that the designers of a video server have to support a large number of languages, but do not want to devote so much RAM to buffers to hold each frame. What other alternatives are available, and what are the advantages and disadvantages of each one?

24. A small video server has eight movies. What does Zipf's law predict as the probabilities for the most popular movie, second most popular movie, and so on down to the least popular movie?

25. A 14-GB disk with 1000 cylinders is used to hold 1000 30-sec MPEG-2 video clips running at 4 Mbps. They are stored according to the organ-pipe algorithm. Assuming Zipf's law, what fraction of the time will the disk arm spend in the middle 10 cylinders?

Spotify USA Inc. Exhibit 1009 Page 213

26. Assuming that the relative demand for films *A*, *B*, *C*, and *D* is described by Zipf's law, what is the expected relative utilization of the four disks in Fig. 7-22 for the four striping methods shown?

27. Two video-on-demand customers started watching the same PAL movie 6 sec apart. If the system speeds up one stream and slows down the other to get them to merge, what percent speed up/down is needed to merge them in 3 min?

28. An MPEG-2 video server uses the round scheme of Fig. 7-24 for NTSC video. All the videos come off a single 10,800 rpm UltraWide SCSI disk with an average seek time of 3 msec. How many streams can be supported?

29. Repeat the previous problem, but now assume that scan-EDF reduces the average seek time by 20%. How many streams can now be supported?

30. Repeat the previous problem once more, but now assume that each frame is striped across four disks, with scan-EDF giving the 20% on each disk. How many streams can now be supported.

31. The text describes using a batch of five data requests to schedule the situation described in Fig. 7-25(a). If all requests take an equal amount of time, what is the maximum time per request allowable in this example?

32. Many of the bitmap images that are supplied for generating computer "wallpaper" use few colors and are easily compressed. A simple compression scheme is the following: choose a data value that does not appear in the input file, and use it as a flag. Read the file, byte by byte, looking for repeated byte values. Copy single values and bytes repeated up to three times directly to the output file. When a repeated string of 4 or more bytes is found, write to the output file a string of three bytes consisting of the flag byte, a byte indicating a count from 4 to 255, and the actual value found in the input file. Write a compression program using this algorithm, and a decompression program that can restore the original file. Extra credit: how can you deal with files that contain the flag byte in their data?

33. Computer animation is accomplished by displaying a sequence of slightly different images. Write a program to calculate the byte by byte difference between two uncompressed bitmap images of the same dimensions. The output will be the same size as the input files, of course. Use this difference file as input to the compression program of the previous problem, and compare the effectiveness of this approach with compression of individual images.

Spotify USA Inc. Exhibit 1009 Page 214

SECOND EDITION



MODERN OPERATING SYSTEMS

Andrew S. Tanenbaum

The widely anticipated revision of this worldwide best seller incorporates the latest developments in operating systems technologies. Hundreds of pages of new material on a wealth of subjects have been added. This authoritative, example-based reference offers practical, hands-on information in constructing and understanding modern operating systems. Continued in this second edition are the "big picture" concepts, presented in the clear and entertaining style that only Andrew S. Tanenbaum can provide. Tanenbaum's long experience as the designer or co-designer of three operating systems brings a knowledge of the subject and wealth of practical detail that few other books can match.

FEATURES

- NEW—New chapters on computer security, multimedia operating systems, and multiple processor systems.
- NEW—Extensive coverage of Linux, UNIX®, and Windows 2000™ as examples.
- NEW—A unique chapter on operating system design not found in other books.
- NEW—Now includes coverage of graphical user interfaces, multiprocessor operating systems, trusted systems, viruses, network terminals, CD-ROM file systems, power management on laptops, RAID, soft timers, stable storage, fair-share scheduling, three-level scheduling, and new paging algorithms.
- NEW—Most chapters have a new section on current research on the chapter's topic.
- NEW—Focus on *single-processor* computer systems; a new book for a follow-up course on distributed systems is also available from Prentice Hall.
- NEW—Over 200 references to books and papers published since the first edition.
- NEW—The Web site for this book contains PowerPoint slides, simulators, figures in various formats, and other teaching aids.

OTHER BEST-SELLING TITLES BY THE AUTHOR AVAILABLE FROM PRENTICE HALL

Distributed Systems by Andrew S. Tanenbaum and Maarten van Steen, 2002

Operating Systems: Design and Implementation 2/e by Andrew S. Tanenbaum and Albert S. Woodhull, 1997

Structured Computer Organization 4/e by Andrew S. Tanenbaum, 1999

Computer Networks 3/e by Andrew S. Tanenbaum, 1996

InformIT
www.informit.com

Pearson Education

Prentice Hall
Upper Saddle River, NJ 07458

http://www.prenhall.com

ISBN 0-13-031358-0
90000
9 780130 313584

Spotify USA Inc. Exhibit 1009 Page 215