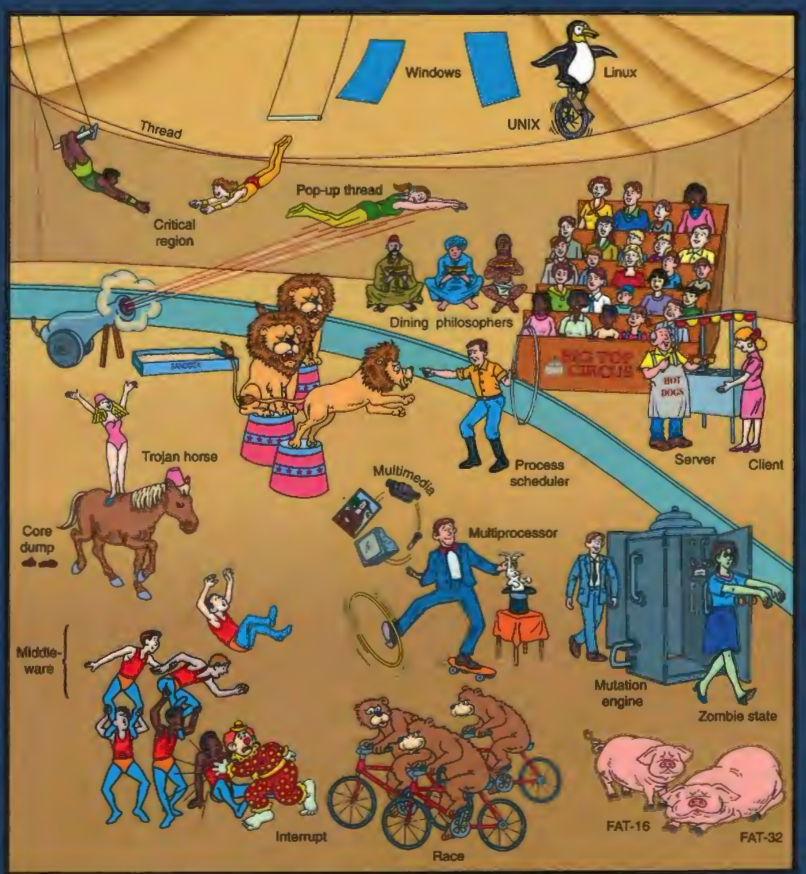
MODERN OPERATING SYSTEMS

SECOND EDITION



Andrew S. Tanenbaum

MODERN
OPERATING SYSTEMS

SECOND EDITION

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.

MODERN OPERATING SYSTEMS

SECOND EDITION

ANDREW S. TANENBAUM

Vrije Universiteit
Amsterdam, The Netherlands

PRENTICE HALL
UPPER SADDLE RIVER, NEW JERSEY 07458

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 4 3 2 1

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*

To Suzanne, Barbara, Marvin, and the memory of Bram and Sweetie π

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

IBM Ex-1011
Page 7

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

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

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

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

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

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

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

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

PREFACE

The world has changed a great deal since the first edition of this book appeared in 1992. Computer networks and distributed systems of all kinds have become very common. Small children now roam the Internet, where previously only computer professionals went. As a consequence, this book has changed a great deal, too.

The most obvious change is that the first edition was about half on single-processor operating systems and half on distributed systems. I chose that format in 1991 because few universities then had courses on distributed systems and whatever students learned about distributed systems had to be put into the operating systems course, for which this book was intended. Now most universities have a separate course on distributed systems, so it is not necessary to try to combine the two subjects into one course and one book. This book is intended for a first course on operating systems, and as such focuses mostly on traditional single-processor systems.

I have coauthored two other books on operating systems. This leads to two possible course sequences.

Practically-oriented sequence:

1. Operating Systems Design and Implementation by Tanenbaum and Woodhull
2. Distributed Systems by Tanenbaum and Van Steen

Traditional sequence:

1. Modern Operating Systems by Tanenbaum
2. Distributed Systems by Tanenbaum and Van Steen

IBM Ex-1011
Page 22

The former sequence uses MINIX and the students are expected to experiment with MINIX in an accompanying laboratory supplementing the first course. The latter sequence does not use MINIX. Instead, some small simulators are available that can be used for student exercises during a first course using this book. These simulators can be found starting on the author's Web page: *www.cs.vu.nl/~ast/* by clicking on Software and supplementary material for my books .

In addition to the major change of switching the emphasis to single-processor operating systems in this book, other major changes include the addition of entire chapters on computer security, multimedia operating systems, and Windows 2000, all important and timely topics. In addition, a new and unique chapter on operating system design has been added.

Another new feature is that many chapters now have a section on research about the topic of the chapter. This is intended to introduce the reader to modern work in processes, memory management, and so on. These sections have numerous references to the current research literature for the interested reader. In addition, Chapter 13 has many introductory and tutorial references.

Finally, numerous topics have been added to this book or heavily revised. These topics include: graphical user intefaces, multiprocessor operating systems, power management for laptops, trusted systems, viruses, network terminals, CD-ROM file systems, mutexes, RAID, soft timers, stable storage, fair-share scheduling, and new paging algorithms. Many new problems have been added and old ones updated. The total number of problems now exceeds 450. A solutions manual is available to professors using this book in a course. They can obtain a copy from their local Prentice Hall representative. In addition, over 250 new references to the current literature have been added to bring the book up to date.

Despite the removal of more than 400 pages of old material, the book has increased in size due to the large amount of new material added. While the book is still suitable for a one-semester or two-quarter course, it is probably too long for a one-quarter or one-trimester course at most universities. For this reason, the book has been designed in a modular way. Any course on operating systems should cover chapters 1 through 6. This is basic material that every student show know.

If additional time is available, additional chapters can be covered. Each of them assumes the reader has finished chapters 1 through 6, but Chaps. 7 through 12 are each self contained, so any desired subset can be used and in any order, depending on the interests of the instructor. In the author's opinion, Chaps. 7 through 12 are much more interesting than the earlier ones. Instructors should tell their students that they have to eat their broccoli before they can have the double chocolate fudge cake dessert.

I would like to thank the following people for their help in reviewing parts of the manuscript: Rida Bazzi, Riccardo Bettati, Felipe Cabrera, Richard Chapman, John Connely, John Dickinson, John Elliott, Deborah Frincke, Chandana Gamage, Robbert Geist, David Golds, Jim Griffioen, Gary Harkin, Frans Kaashoek, Muk-

kai Krishnamoorthy, Monica Lam, Jussi Leiwo, Herb Mayer, Kirk McKusick, Evi Nemeth, Bill Potvin, Prasant Shenoy, Thomas Skinner, Xian-He Sun, William Terry, Robbert Van Renesse, and Maarten van Steen. Jamie Hanrahan, Mark Russinovich, and Dave Solomon were enormously knowledgeable about Windows 2000 and very helpful. Special thanks go to Al Woodhull for valuable reviews and thinking of many new end-of-chapter problems.

My students were also helpful with comments and feedback, especially Staas de Jong, Jan de Vos, Niels Drost, David Fokkema, Auke Folkerts, Peter Groenewegen, Wilco Ibes, Stefan Jansen, Jeroen Ketema, Joeri Mulder, Irwin Oppenheim, Stef Post, Umar Rehman, Daniel Rijkhof, Maarten Sander, Maurits van der Schee, Rik van der Stoel, Mark van Driel, Dennis van Veen, and Thomas Zeeman.

Barbara and Marvin are still wonderful, as usual, each in a unique way. Finally, last but not least, I would like to thank Suzanne for her love and patience, not to mention all the *druiven* and *kersen*, which have replaced the *sinasappelsap* in recent times.

Andrew S. Tanenbaum

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.

IBM Ex-1011
Page 25

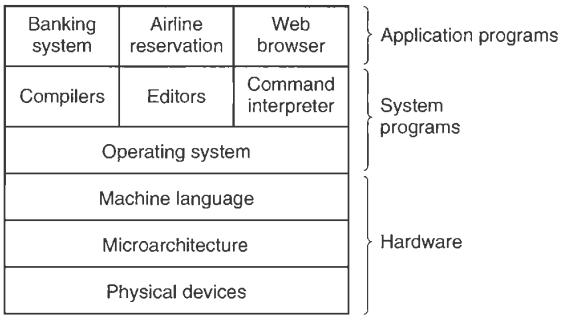


Figure 1-1. A computer system consists of hardware, system programs, and application programs.

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

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.

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,

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

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.

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

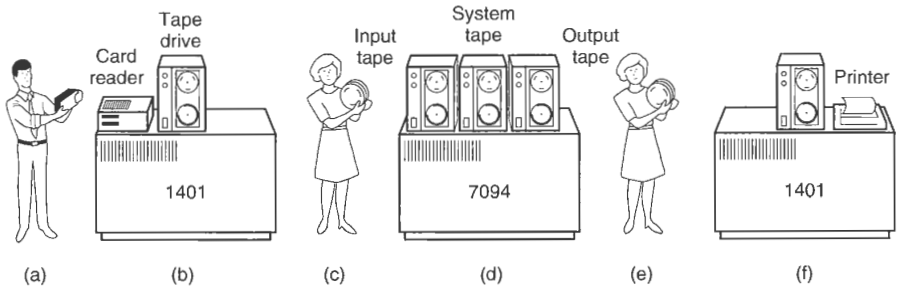


Figure 1-2. An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

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.

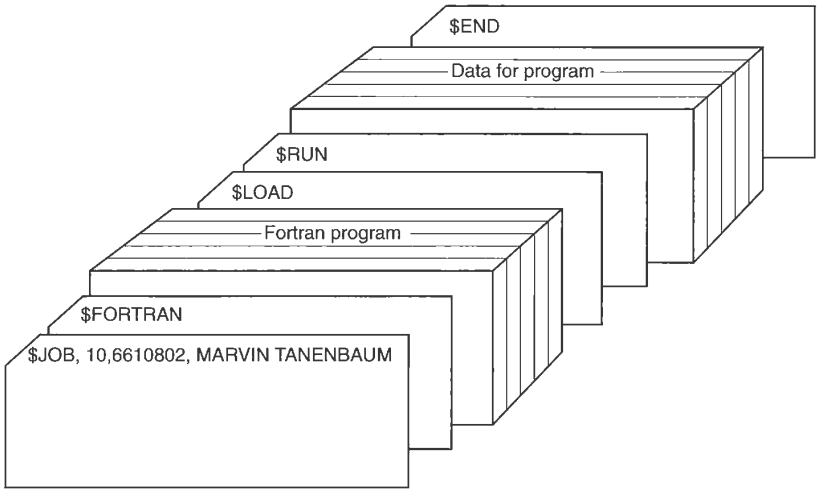


Figure 1-3. Structure of a typical FMS job.

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.

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

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.

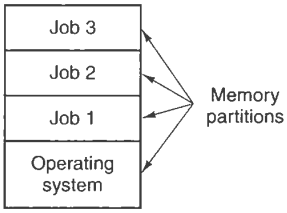


Figure 1-4. A multiprogramming system with three jobs in memory.

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.

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

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

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

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

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

(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,

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.

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.

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.

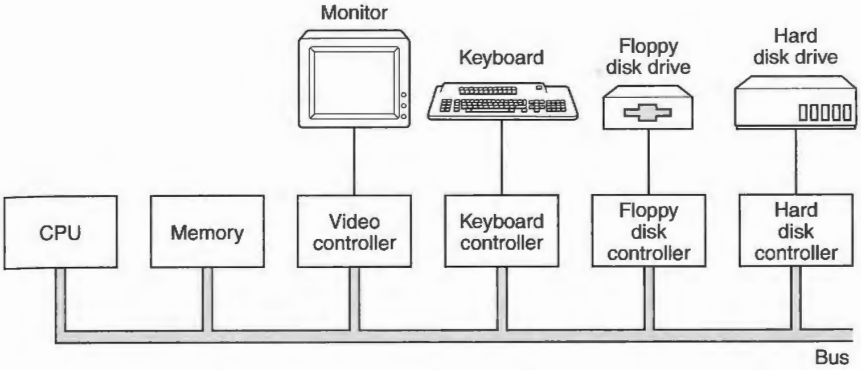


Figure 1-5. Some of the components of a simple personal computer.

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

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.

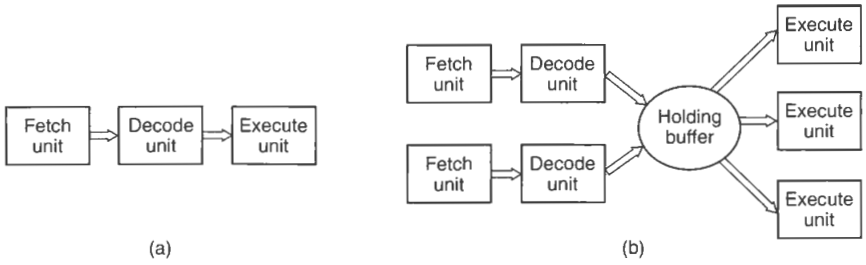


Figure 1-6. (a) A three-stage pipeline. (b) A superscalar CPU.

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

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**

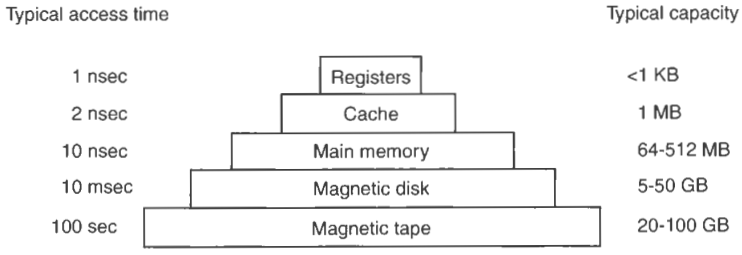


Figure 1-7. A typical memory hierarchy. The numbers are very rough approximations.

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.

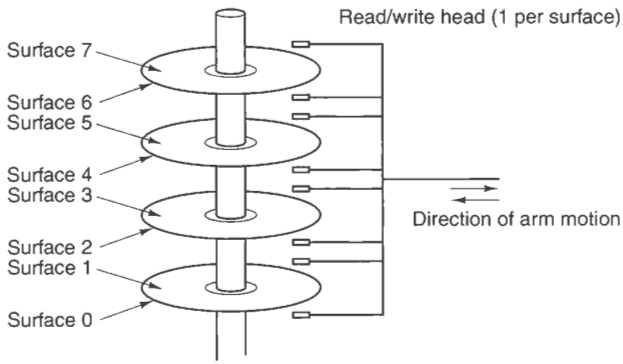


Figure 1-8. Structure of a disk drive.

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

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-

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).

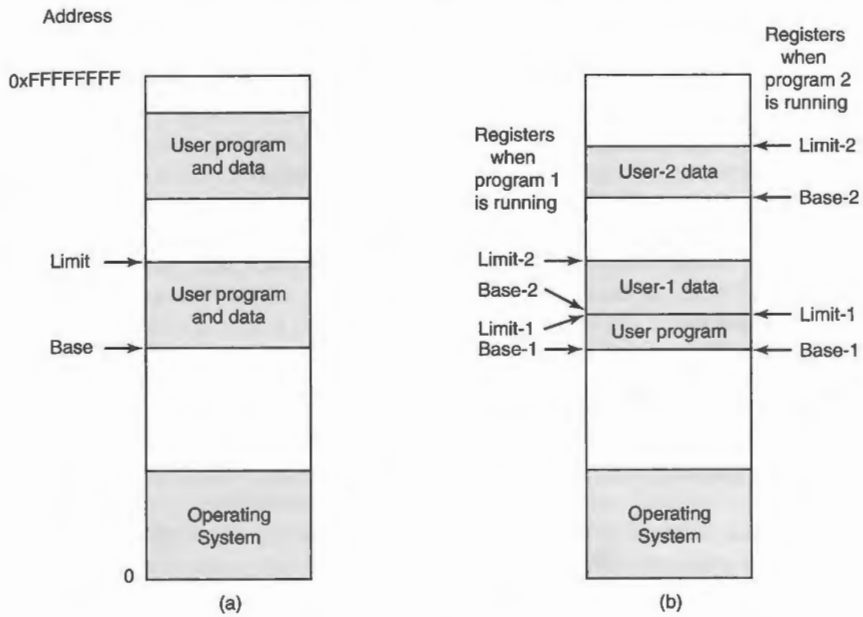


Figure 1-9. (a) Use of one base-limit pair. The program can access memory between the base and the limit. (b) Use of two base-limit pairs. The program code is between Base-1 and Limit-1 whereas the data are between Base-2 and Limit-2.

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

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

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

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).

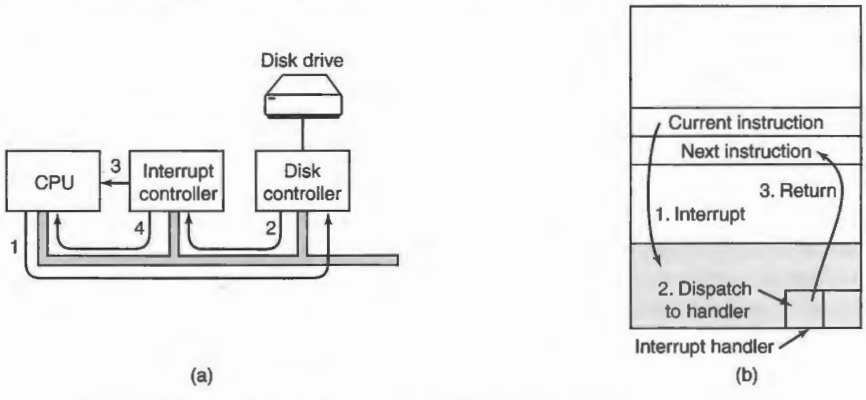


Figure 1-10. (a) The steps in starting an I/O device and getting an interrupt. (b) Interrupt processing involves taking the interrupt, running the interrupt handler, and returning to the user program.

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).

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-

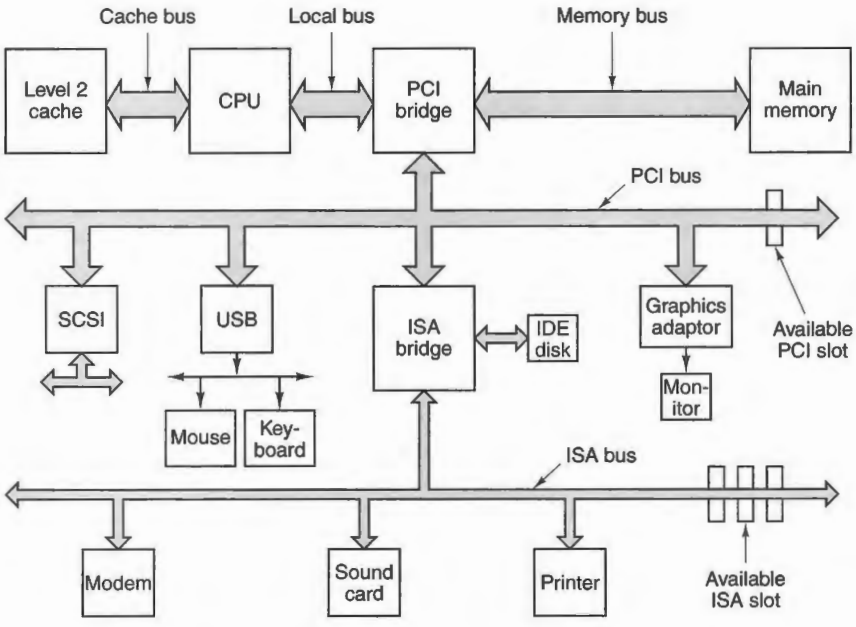


Figure 1-11. The structure of a large Pentium system

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

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

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

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.

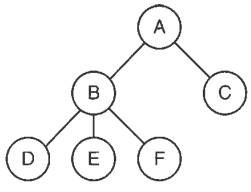


Figure 1-12. A process tree. Process *A* created two child processes, *B* and *C*. Process *B* created three child processes, *D*, *E*, and *F*.

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.

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

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.

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,

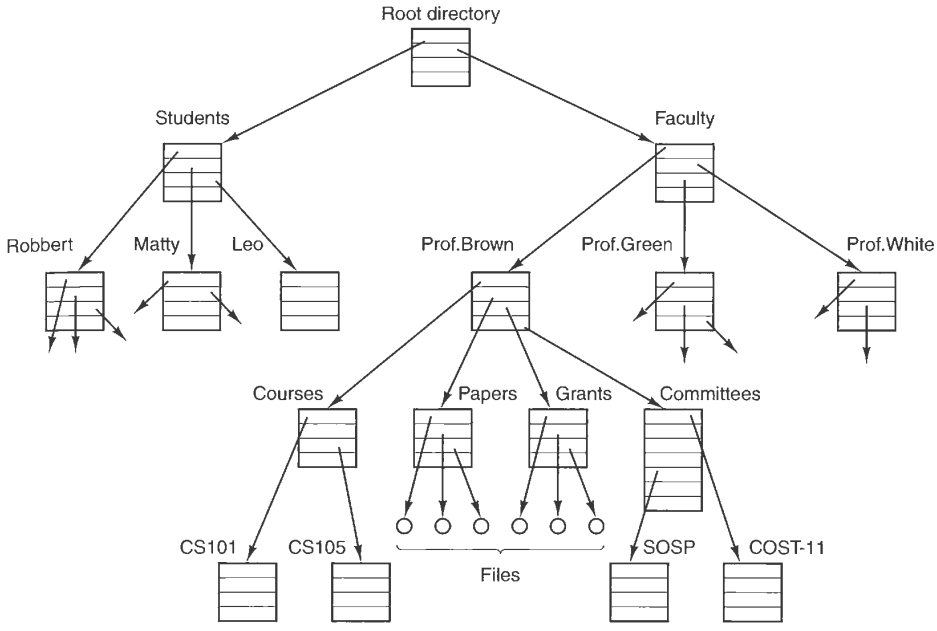


Figure 1-14. A file system for a university department.

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

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.

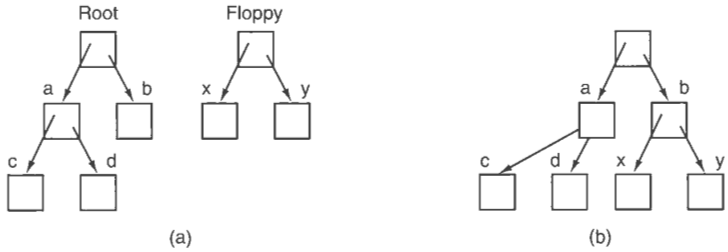


Figure 1-15. (a) Before mounting, the files on drive 0 are not accessible. (b) After mounting, they are part of the file hierarchy.

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

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.

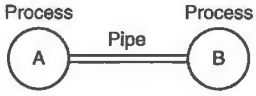


Figure 1-16. Two processes connected by a pipe.

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-

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).

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

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

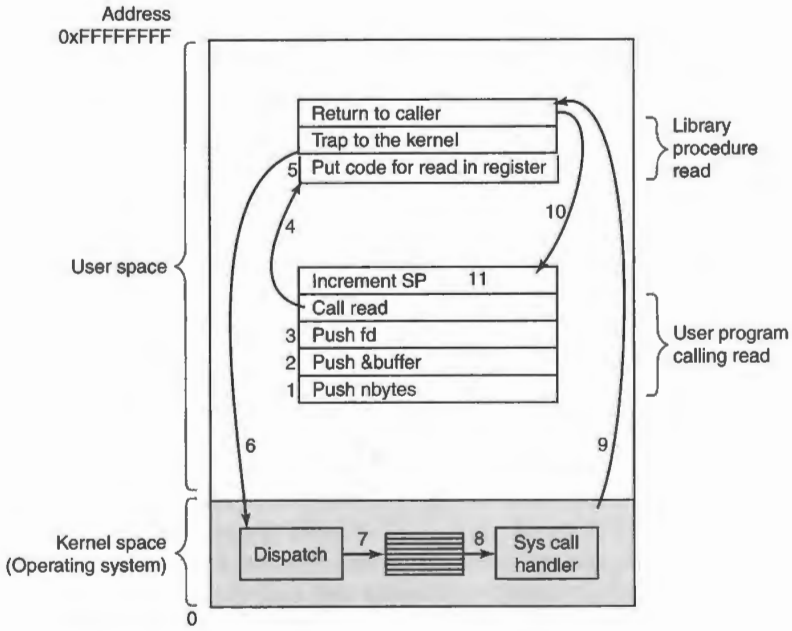


Figure 1-17. The 11 steps in making the system call read(fd, buffer, nbytes).

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.

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

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

```
#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.

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.

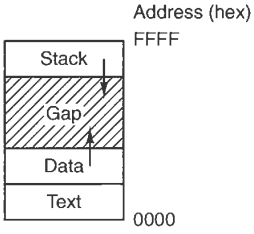


Figure 1-20. Processes have three segments: text, data, and stack.

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.

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

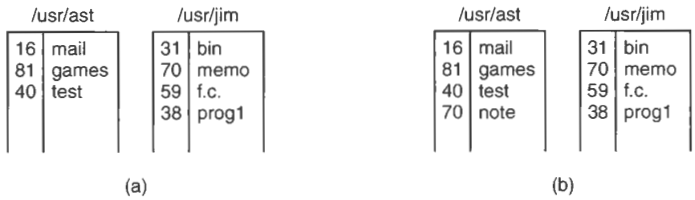


Figure 1-21. (a) Two directories before linking */usr/jim/memo* to ast's directory. (b) The same directories after linking.

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.

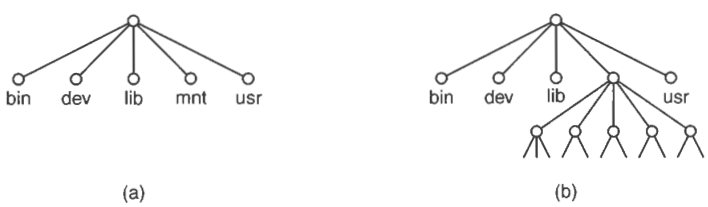


Figure 1-22. (a) File system before the mount. (b) File system after the mount.

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

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).

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.

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

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.

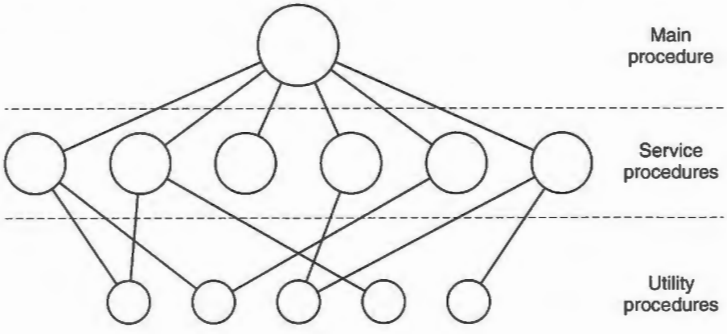


Figure 1-24. A simple structuring model for a monolithic system.

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

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

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.

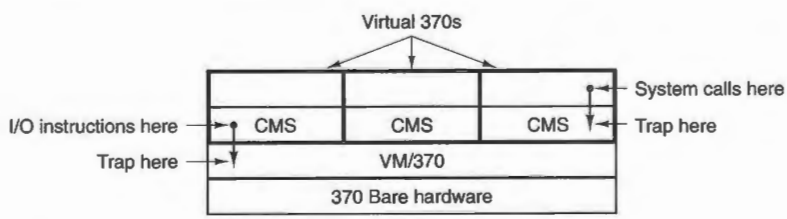


Figure 1-26. The structure of VM/370 with CMS.

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.

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

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

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.

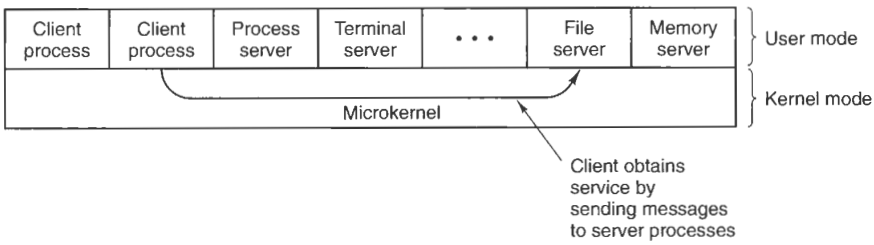


Figure 1-27. The client-server model.

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

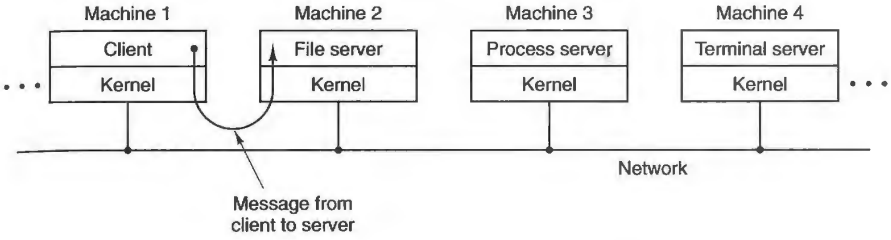


Figure 1-28. The client-server model in a distributed system.

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.

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

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-

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

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?

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?

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

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.

2

PROCESSES AND THREADS

We are now about to embark on a detailed study of how operating systems are designed and constructed. The most central concept in any operating system is the *process*: an abstraction of a running program. Everything else hinges on this concept, and it is important that the operating system designer (and student) have a thorough understanding of what a process is as early as possible.

2.1 PROCESSES

All modern computers can do several things at the same time. While running a user program, a computer can also be reading from a disk and outputting text to a screen or printer. In a multiprogramming system, the CPU also switches from program to program, running each for tens or hundreds of milliseconds. While, strictly speaking, at any instant of time, the CPU is running only one program, in the course of 1 second, it may work on several programs, thus giving the users the illusion of parallelism. Sometimes people speak of **pseudoparallelism** in this context, to contrast it with the true hardware parallelism of **multiprocessor** systems (which have two or more CPUs sharing the same physical memory). Keeping track of multiple, parallel activities is hard for people to do. Therefore, operating system designers over the years have evolved a conceptual model (sequential processes) that makes parallelism easier to deal with. That model, its uses, and some of its consequences form the subject of this chapter.

IBM Ex-1011
Page 95

2.1.1 The Process Model

In this model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of **sequential processes**, or just **processes** for short. A process is just an executing program, including the current values of the program counter, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, of course, the real CPU switches back and forth from process to process, but to understand the system, it is much easier to think about a collection of processes running in (pseudo) parallel, than to try to keep track of how the CPU switches from program to program. This rapid switching back and forth is called **multiprogramming**, as we saw in Chap. 1.

In Fig. 2-1(a) we see a computer multiprogramming four programs in memory. In Fig. 2-1(b) we see four processes, each with its own flow of control (i.e., its own logical program counter), and each one running independently of the other ones. Of course, there is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter. When it is finished for the time being, the physical program counter is saved in the process' logical program counter in memory. In Fig. 2-1(c) we see that viewed over a long enough time interval, all the processes have made progress, but at any given instant only one process is actually running.

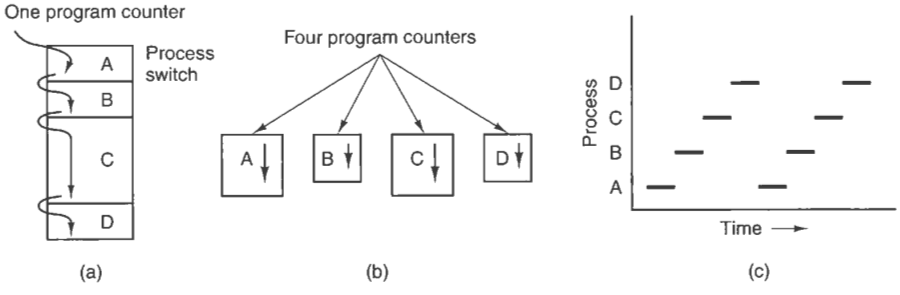


Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform and probably not even reproducible if the same processes are run again. Thus, processes must not be programmed with built-in assumptions about timing. Consider, for example, an I/O process that starts a streamer tape to restore backed up files, executes an idle loop 10,000 times to let it get up to speed, and then issues a command to read the first record. If the CPU decides to switch to another process during the idle loop, the tape process might not run again until after the first record was already past the read head. When a process has critical real-time requirements like this,

that is, particular events *must* occur within a specified number of milliseconds, special measures must be taken to ensure that they do occur. Normally, however, most processes are not affected by the underlying multiprogramming of the CPU or the relative speeds of different processes.

The difference between a process and a program is subtle, but crucial. An analogy make help here. Consider a culinary-minded computer scientist who is baking a birthday cake for his daughter. He has a birthday cake recipe and a kitchen well stocked with all the input: flour, eggs, sugar, extract of vanilla, and so on. In this analogy, the recipe is the program (i.e., an algorithm expressed in some suitable notation), the computer scientist is the processor (CPU), and the cake ingredients are the input data. The process is the activity consisting of our baker reading the recipe, fetching the ingredients, and baking the cake.

Now imagine that the computer scientist's son comes running in crying, saying that he has been stung by a bee. The computer scientist records where he was in the recipe (the state of the current process is saved), gets out a first aid book, and begins following the directions in it. Here we see the processor being switched from one process (baking) to a higher-priority process (administering medical care), each having a different program (recipe versus first aid book). When the bee sting has been taken care of, the computer scientist goes back to his cake, continuing at the point where he left off.

The key idea here is that a process is an activity of some kind. It has a program, input, output, and a state. A single processor may be shared among several processes, with some scheduling algorithm being used to determine when to stop work on one process and service a different one.

2.1.2 Process Creation

Operating systems need some way to make sure all the necessary processes exist. In very simple systems, or in systems designed for running only a single application (e.g., the controller in a microwave oven), it may be possible to have all the processes that will ever be needed be present when the system comes up. In general-purpose systems, however, some way is needed to create and terminate processes as needed during operation. We will now look at some of the issues.

There are four principal events that cause processes to be created:

1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

When an operating system is booted, typically several processes are created. Some of these are foreground processes, that is, processes that interact with

(human) users and perform work for them. Others are background processes, which are not associated with particular users, but instead have some specific function. For example, one background process may be designed to accept incoming email, sleeping most of the day but suddenly springing to life when email arrives. Another background process may be designed to accept incoming requests for Web pages hosted on that machine, waking up when a request arrives to service the request. Processes that stay in the background to handle some activity such as email, Web pages, news, printing, and so on are called **daemons**. Large systems commonly have dozens of them. In UNIX, the *ps* program can be used to list the running processes. In Windows 95/98/Me, typing CTRL-ALT-DEL once shows what's running. In Windows 2000, the task manager is used.

In addition to the processes created at boot time, new processes can be created afterward as well. Often a running process will issue system calls to create one or more new processes to help it do its job. Creating new processes is particularly useful when the work to be done can easily be formulated in terms of several related, but otherwise independent interacting processes. For example, if a large amount of data is being fetched over a network for subsequent processing, it may be convenient to create one process to fetch the data and put them in a shared buffer while a second process removes the data items and processes them. On a multiprocessor, allowing each process to run on a different CPU may also make the job go faster.

In interactive systems, users can start a program by typing a command or (double) clicking an icon. Taking either of these actions starts a new process and runs the selected program in it. In command-based UNIX systems running X Windows, the new process takes over the window in which it was started. In Microsoft Windows, when a process is started it does not have a window, but it can create one (or more) and most do. In both systems, users may have multiple windows open at once, each running some process. Using the mouse, the user can select a window and interact with the process, for example, providing input when needed.

The last situation in which processes are created applies only to the batch systems found on large mainframes. Here users can submit batch jobs to the system (possibly remotely). When the operating system decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

Technically, in all these cases, a new process is created by having an existing process execute a process creation system call. That process may be a running user process, a system process invoked from the keyboard or mouse, or a batch manager process. What that process does is execute a system call to create the new process. This system call tells the operating system to create a new process and indicates, directly or indirectly, which program to run in it.

In UNIX, there is only one system call to create a new process: fork. This call creates an exact clone of the calling process. After the fork, the two processes, the

parent and the child, have the same memory image, the same environment strings, and the same open files. That is all there is. Usually, the child process then executes execve or a similar system call to change its memory image and run a new program. For example, when a user types a command, say, *sort*, to the shell, the shell forks off a child process and the child executes *sort*. The reason for this two-step process is to allow the child to manipulate its file descriptors after the fork but before the execve to accomplish redirection of standard input, standard output, and standard error.

In Windows, in contrast, a single Win32 function call, CreateProcess, handles both process creation and loading the correct program into the new process. This call has 10 parameters, which include the program to be executed, the command line parameters to feed that program, various security attributes, bits that control whether open files are inherited, priority information, a specification of the window to be created for the process (if any), and a pointer to a structure in which information about the newly created process is returned to the caller. In addition to CreateProcess, Win32 has about 100 other functions for managing and synchronizing processes and related topics.

In both UNIX and Windows, after a process is created, both the parent and child have their own distinct address spaces. If either process changes a word in its address space, the change is not visible to the other process. In UNIX, the child's initial address space is a *copy* of the parent's, but there are two distinct address spaces involved; no writable memory is shared (some UNIX implementations share the program text between the two since that cannot be modified). It is, however, possible for a newly created process to share some of its creator's other resources, such as open files. In Windows, the parent's and child's address spaces are different from the start.

2.1.3 Process Termination

After a process has been created, it starts running and does whatever its job is. However, nothing lasts forever, not even processes. Sooner or later the new process will terminate, usually due to one of the following conditions:

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Most processes terminate because they have done their work. When a compiler has compiled the program given to it, the compiler executes a system call to tell the operating system that it is finished. This call is exit in UNIX and ExitProcess in Windows. Screen-oriented programs also support voluntary termination.

Word processors, Internet browsers and similar programs always have an icon or menu item that the user can click to tell the process to remove any temporary files it has open and then terminate.

The second reason for termination is that the process discovers a fatal error. For example, if a user types the command

```
cc foo.c
```

to compile the program *foo.c* and no such file exists, the compiler simply exits. Screen-oriented interactive processes generally do not exit when given bad parameters. Instead they pop up a dialog box and ask the user to try again.

The third reason for termination is an error caused by the process, often due to a program bug. Examples include executing an illegal instruction, referencing nonexistent memory, or dividing by zero. In some systems (e.g., UNIX), a process can tell the operating system that it wishes to handle certain errors itself, in which case the process is signaled (interrupted) instead of terminated when one of the errors occurs.

The fourth reason a process might terminate is that a process executes a system call telling the operating system to kill some other process. In UNIX this call is kill. The corresponding Win32 function is TerminateProcess. In both cases, the killer must have the necessary authorization to do in the killee. In some systems, when a process terminates, either voluntarily or otherwise, all processes it created are immediately killed as well. Neither UNIX nor Windows works this way, however.

2.1.4 Process Hierarchies

In some systems, when a process creates another process, the parent process and child process continue to be associated in certain ways. The child process can itself create more processes, forming a process hierarchy. Note that unlike plants and animals that use sexual reproduction, a process has only one parent (but zero, one, two, or more children).

In UNIX, a process and all of its children and further descendants together form a process group. When a user sends a signal from the keyboard, the signal is delivered to all members of the process group currently associated with the keyboard (usually all active processes that were created in the current window). Individually, each process can catch the signal, ignore the signal, or take the default action, which is to be killed by the signal.

As another example of where the process hierarchy plays a role, let us look at how UNIX initializes itself when it is started. A special process, called *init*, is present in the boot image. When it starts running, it reads a file telling how many terminals there are. Then it forks off one new process per terminal. These processes wait for someone to log in. If a login is successful, the login process executes a shell to accept commands. These commands may start up more processes,

and so forth. Thus, all the processes in the whole system belong to a single tree, with *init* at the root.

In contrast, Windows does not have any concept of a process hierarchy. All processes are equal. The only place where there is something like a process hierarchy is that when a process is created, the parent is given a special token (called a **handle**) that it can use to control the child. However, it is free to pass this token to some other process, thus invalidating the hierarchy. Processes in UNIX cannot disinherit their children.

2.1.5 Process States

Although each process is an independent entity, with its own program counter and internal state, processes often need to interact with other processes. One process may generate some output that another process uses as input. In the shell command

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

the first process, running *cat*, concatenates and outputs three files. The second process, running *grep*, selects all lines containing the word "tree." Depending on the relative speeds of the two processes (which depends on both the relative complexity of the programs and how much CPU time each one has had), it may happen that *grep* is ready to run, but there is no input waiting for it. It must then block until some input is available.

When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not yet available. It is also possible for a process that is conceptually ready and able to run to be stopped because the operating system has decided to allocate the CPU to another process for a while. These two conditions are completely different. In the first case, the suspension is inherent in the problem (you cannot process the user's command line until it has been typed). In the second case, it is a technicality of the system (not enough CPUs to give each process its own private processor). In Fig. 2-2 we see a state diagram showing the three states a process may be in:

1. Running (actually using the CPU at that instant).
2. Ready (runnable; temporarily stopped to let another process run).
3. Blocked (unable to run until some external event happens).

Logically, the first two states are similar. In both cases the process is willing to run, only in the second one, there is temporarily no CPU available for it. The third state is different from the first two in that the process cannot run, even if the CPU has nothing else to do.

Four transitions are possible among these three states, as shown. Transition 1 occurs when a process discovers that it cannot continue. In some systems the

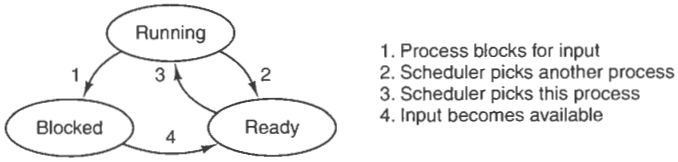


Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

process must execute a system call, such as block or pause, to get into blocked state. In other systems, including UNIX, when a process reads from a pipe or special file (e.g., a terminal) and there is no input available, the process is automatically blocked.

Transitions 2 and 3 are caused by the process scheduler, a part of the operating system, without the process even knowing about them. Transition 2 occurs when the scheduler decides that the running process has run long enough, and it is time to let another process have some CPU time. Transition 3 occurs when all the other processes have had their fair share and it is time for the first process to get the CPU to run again. The subject of scheduling, that is, deciding which process should run when and for how long, is an important one; we will look at it later in this chapter. Many algorithms have been devised to try to balance the competing demands of efficiency for the system as a whole and fairness to individual processes. We will study some of them later in this chapter.

Transition 4 occurs when the external event for which a process was waiting (such as the arrival of some input) happens. If no other process is running at that instant, transition 3 will be triggered and the process will start running. Otherwise it may have to wait in *ready* state for a little while until the CPU is available and its turn comes.

Using the process model, it becomes much easier to think about what is going on inside the system. Some of the processes run programs that carry out commands typed in by a user. Other processes are part of the system and handle tasks such as carrying out requests for file services or managing the details of running a disk or a tape drive. When a disk interrupt occurs, the system makes a decision to stop running the current process and run the disk process, which was blocked waiting for that interrupt. Thus, instead of thinking about interrupts, we can think about user processes, disk processes, terminal processes, and so on, which block when they are waiting for something to happen. When the disk has been read or the character typed, the process waiting for it is unblocked and is eligible to run again.

This view gives rise to the model shown in Fig. 2-3. Here the lowest level of the operating system is the scheduler, with a variety of processes on top of it. All the interrupt handling and details of actually starting and stopping processes are hidden away in what is here called the scheduler, which is actually not much

code. The rest of the operating system is nicely structured in process form. Few real systems are as nicely structured as this, however.

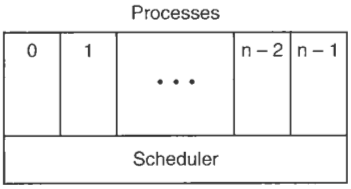


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

2.1.6 Implementation of Processes

To implement the process model, the operating system maintains a table (an array of structures), called the **process table**, with one entry per process. (Some authors call these entries **process control blocks**.) This entry contains information about the process' state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from *running* to *ready* or *blocked* state so that it can be restarted later as if it had never been stopped.

Figure 2-4 shows some of the more important fields in a typical system. The fields in the first column relate to process management. The other two columns relate to memory management and file management, respectively. It should be noted that precisely which fields the process table has is highly system dependent, but this figure gives a general idea of the kinds of information needed.

Now that we have looked at the process table, it is possible to explain a little more about how the illusion of multiple sequential processes is maintained on a machine with one CPU and many I/O devices. Associated with each I/O device class (e.g., floppy disks, hard disks, timers, terminals) is a location (often near the bottom of memory) called the **interrupt vector**. It contains the address of the interrupt service procedure. Suppose that user process 3 is running when a disk interrupt occurs. User process 3's program counter, program status word, and possibly one or more registers are pushed onto the (current) stack by the interrupt hardware. The computer then jumps to the address specified in the disk interrupt vector. That is all the hardware does. From here on, it is up to the software, in particular, the interrupt service procedure.

All interrupts start by saving the registers, often in the process table entry for the current process. Then the information pushed onto the stack by the interrupt is removed and the stack pointer is set to point to a temporary stack used by the

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure 2-4. Some of the fields of a typical process table entry.

process handler. Actions such as saving the registers and setting the stack pointer cannot even be expressed in high-level languages such as C, so they are performed by a small assembly language routine, usually the same one for all interrupts since the work of saving the registers is identical, no matter what the cause of the interrupt is.

When this routine is finished, it calls a C procedure to do the rest of the work for this specific interrupt type. (We assume the operating system is written in C, the usual choice for all real operating systems.) When it has done its job, possibly making some process now ready, the scheduler is called to see who to run next. After that, control is passed back to the assembly language code to load up the registers and memory map for the now-current process and start it running. Interrupt handling and scheduling are summarized in Fig. 2-5. It is worth noting that the details vary somewhat from system to system.

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

2.2 THREADS

In traditional operating systems, each process has an address space and a single thread of control. In fact, that is almost the definition of a process. Nevertheless, there are frequently situations in which it is desirable to have multiple threads of control in the same address space running in quasi-parallel, as though they were separate processes (except for the shared address space). In the following sections we will discuss these situations and their implications.

2.2.1 The Thread Model

The process model as we have discussed it thus far is based on two independent concepts: resource grouping and execution. Sometimes it is useful to separate them; this is where threads come in.

One way of looking at a process is that it is way to group related resources together. A process has an address space containing program text and data, as well as other resources. These resource may include open files, child processes, pending alarms, signal handlers, accounting information, and more. By putting them together in the form of a process, they can be managed more easily.

The other concept a process has is a thread of execution, usually shortened to just **thread**. The thread has a program counter that keeps track of which instruction to execute next. It has registers, which hold its current working variables. It has a stack, which contains the execution history, with one frame for each procedure called but not yet returned from. Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately. Processes are used to group resources together; threads are the entities scheduled for execution on the CPU.

What threads add to the process model is to allow multiple executions to take place in the same process environment, to a large degree independent of one another. Having multiple threads running in parallel in one process is analogous to having multiple processes running in parallel in one computer. In the former case, the threads share an address space, open files, and other resources. In the latter case, processes share physical memory, disks, printers, and other resources. Because threads have some of the properties of processes, they are sometimes called **lightweight processes**. The term **multithreading** is also used to describe the situation of allowing multiple threads in the same process.

In Fig. 2-6(a) we see three traditional processes. Each process has its own address space and a single thread of control. In contrast, in Fig. 2-6(b) we see a single process with three threads of control. Although in both cases we have three threads, in Fig. 2-6(a) each of them operates in a different address space, whereas in Fig. 2-6(b) all three of them share the same address space.

When a multithreaded process is run on a single-CPU system, the threads take turns running. In Fig. 2-1, we saw how multiprogramming of processes works.

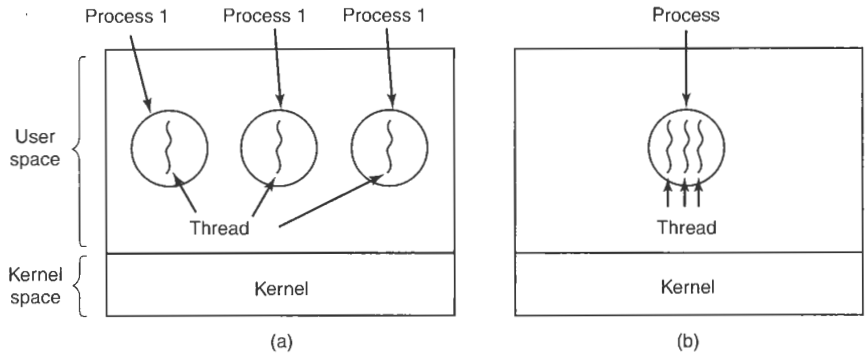


Figure 2-6. (a) Three processes each with one thread. (b) One process with three threads.

By switching back and forth among multiple processes, the system gives the illusion of separate sequential processes running in parallel. Multithreading works the same way. The CPU switches rapidly back and forth among the threads providing the illusion that the threads are running in parallel, albeit on a slower CPU than the real one. With three compute-bound threads in a process, the threads would appear to be running in parallel, each one on a CPU with one-third the speed of the real CPU.

Different threads in a process are not quite as independent as different processes. All threads have exactly the same address space, which means that they also share the same global variables. Since every thread can access every memory address within the process' address space, one thread can read, write, or even completely wipe out another thread's stack. There is no protection between threads because (1) it is impossible, and (2) it should not be necessary. Unlike different processes, which may be from different users and which may be hostile to one another, a process is always owned by a single user, who has presumably created multiple threads so that they can cooperate, not fight. In addition to sharing an address space, all the threads share the same set of open files, child processes, alarms, and signals, etc. as shown in Fig. 2-7. Thus the organization of Fig. 2-6(a) would be used when the three processes are essentially unrelated, whereas Fig. 2-6(b) would be appropriate when the three threads are actually part of the same job and are actively and closely cooperating with each other.

The items in the first column are process properties, not thread properties. For example, if one thread opens a file, that file is visible to the other threads in the process and they can read and write it. This is logical since the process is the unit of resource management, not the thread. If each thread had its own address space, open files, pending alarms, and so on, it would be a separate process. What we are trying to achieve with the thread concept is the ability for multiple threads

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-7. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

of execution to share a set of resources so they can work together closely to perform some task.

Like a traditional process (i.e., a process with only one thread), a thread can be in any one of several states: running, blocked, ready, or terminated. A running thread currently has the CPU and is active. A blocked thread is waiting for some event to unblock it. For example, when a thread performs a system call to read from the keyboard, it is blocked until input is typed. A thread can block waiting for some external event to happen or for some other thread to unblock it. A ready thread is scheduled to run and will as soon as its turn comes up. The transitions between thread states are the same as the transitions between process states and are illustrated in Fig. 2-2.

It is important to realize that each thread has its own stack, as shown in Fig. 2-8. Each thread's stack contains one frame for each procedure called but not yet returned from. This frame contains the procedure's local variables and the return address to use when the procedure call has finished. For example, if procedure *X* calls procedure *Y* and this one calls procedure *Z*, while *Z* is executing the frames for *X*, *Y*, and *Z* will all be on the stack. Each thread will generally call different procedures and a thus a different execution history. This is why is thread needs its own stack.

When multithreading is present, processes normally start with a single thread present. This thread has the ability to create new threads by calling a library procedure, for example, *thread_create*. A parameter to *thread_create* typically specifies the name of a procedure for the new thread to run. It is not necessary (or even possible) to specify anything about the new thread's address space since it automatically runs in the address space of the creating thread. Sometimes threads are hierarchical, with a parent-child relationship, but often no such relationship exists, with all threads being equal. With or without a hierarchical relationship, the creating thread is usually returned a thread identifier that names the new thread.

When a thread has finished its work, it can exit by calling a library procedure, say, *thread_exit*. It then vanishes and is no longer schedulable. In some thread

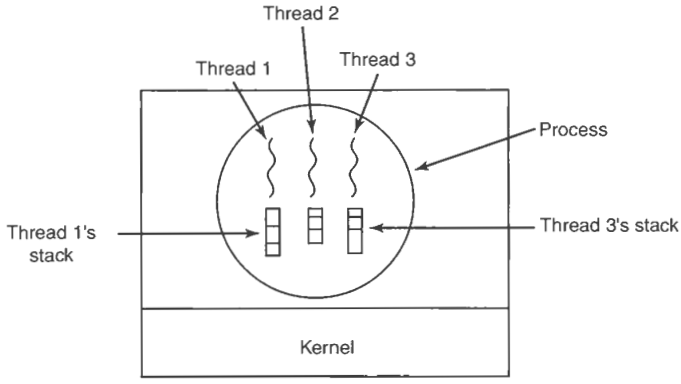


Figure 2-8. Each thread has its own stack.

systems, one thread can wait for a (specific) thread to exit by calling a procedure, for example, *thread_wait*. This procedure blocks the calling thread until a (specific) thread has exited. In this regard, thread creation and termination is very much like process creation and termination, with approximately the same options as well.

Another common thread call is *thread_yield*, which allows a thread to voluntarily give up the CPU to let another thread run. Such a call is important because there is no clock interrupt to actually enforce timesharing as there is with processes. Thus it is important for threads to be polite and voluntarily surrender the CPU from time to time to give other threads a chance to run. Other calls allow one thread to wait for another thread to finish some work, for a thread to announce that it has finished some work, and so on.

While threads are often useful, they also introduce a number of complications into the programming model. To start with, consider the effects of the UNIX fork system call. If the parent process has multiple threads, should the child also have them? If not, the process may not function properly, since all of them may be essential.

However, if the child process gets as many threads as the parent, what happens if a thread in the parent was blocked on a read call, say, from the keyboard? Are two threads now blocked on the keyboard, one in the parent and one in the child? When a line is typed, do both threads get a copy of it? Only the parent? Only the child? The same problem exists with open network connections.

Another class of problems is related to the fact that threads share many data structures. What happens if one thread closes a file while another one is still reading from it? Suppose that one thread notices that there is too little memory and starts allocating more memory. Part way through, a thread switch occurs, and the new thread also notices that there is too little memory and also starts allocating

more memory. Memory will probably be allocated twice. These problems can be solved with some effort, but careful thought and design are needed to make multithreaded programs work correctly.

2.2.2 Thread Usage

Having described what threads are, it is now time to explain why anyone wants them. The main reason for having threads is that in many applications, multiple activities are going on at once. Some of these may block from time to time. By decomposing such an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.

We have seen this argument before. It is precisely the argument for having processes. Instead of thinking about interrupts, timers, and context switches, we can think about parallel processes. Only now with threads we add a new element: the ability for the parallel entities to share an address space and all of its data among themselves. This ability is essential for certain applications, which is why having multiple processes (with their separate address spaces) will not work.

A second argument for having threads is that since they do not have any resources attached to them, they are easier to create and destroy than processes. In many systems, creating a thread goes 100 times faster than creating a process. When the number of threads needed changes dynamically and rapidly, this property is useful.

A third reason for having threads is also a performance argument. Threads yield no performance gain when all of them are CPU bound, but when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, thus speeding up the application.

Finally, threads are useful on systems with multiple CPUs, where real parallelism is possible. We will come back to this issue in Chap. 8.

It is probably easiest to see why threads are useful by giving some concrete examples. As a first example, consider a word processor. Most word processors display the document being created on the screen formatted exactly as it will appear on the printed page. In particular, all the line breaks and page breaks are in their correct and final position so the user can inspect them and change the document if need be (e.g., to eliminate widows and orphans—incomplete top and bottom lines on a page, which are considered esthetically unpleasing).

Suppose that the user is writing a book. From the author's point of view, it is easiest to keep the entire book as a single file to make it easier to search for topics, perform global substitutions, and so on. Alternatively, each chapter might be a separate file. However, having every section and subsection as a separate file is a real nuisance when global changes have to be made to the entire book since then hundreds of files have to be individually edited. For example, if proposed standard xxxx is approved just before the book goes to press, all occurrences of "Draft Standard xxxx" have to be changed to "Standard xxxx" at

the last minute. If the entire book is one file, typically a single command can do all the substitutions. In contrast, if the book is spread over 300 files, each one must be edited separately.

Now consider what happens when the user suddenly deletes one sentence from page 1 of an 800-page document. After checking the changed page to make sure it is correct, the user now wants to make another change on page 600 and types in a command telling the word processor to go to that page (possibly by searching for a phrase occurring only there). The word processor is now forced to reformat the entire book up to page 600 on the spot because it does not know what the first line of page 600 will be until it has processed all the previous pages. There may be a substantial delay before page 600 can be displayed, leading to an unhappy user.

Threads can help here. Suppose that the word processor is written as a two-threaded program. One thread interacts with the user and the other handles reformatting in the background. As soon as the sentence is deleted from page 1, the interactive thread tells the reformatting thread to reformat the whole book. Meanwhile, the interactive thread continues to listen to the keyboard and mouse and responds to simple commands like scrolling page 1 while the other thread is computing madly in the background. With a little luck, the reformatting will be completed before the user asks to see page 600, so it can be displayed instantly.

While we are at it, why not add a third thread? Many word processors have a feature of automatically saving the entire file to disk every few minutes to protect the user against losing a day's work in the event of a program crash, system crash, or power failure. The third thread can handle the disk backups without interfering with the other two. The situation with three threads is shown in Fig. 2-9.

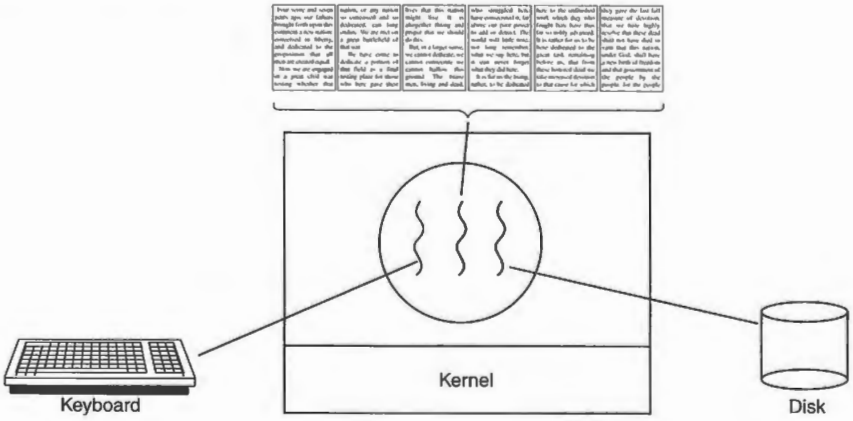


Figure 2-9. A word processor with three threads.

If the program were single-threaded, then whenever a disk backup started, commands from the keyboard and mouse would be ignored until the backup was finished. The user would perceive this as sluggish performance. Alternatively, keyboard and mouse events could interrupt the disk backup, allowing good performance but leading to a complex interrupt-driven programming model. With three threads, the programming model is much simpler. The first thread just interacts with the user. The second thread reformats the document when told to. The third thread writes the contents of RAM to disk periodically.

It should be clear that having three separate processes would not work here because all three threads need to operate on the document. By having three threads instead of three processes, they share a common memory and thus all have access to the document being edited.

An analogous situation exists with many other interactive programs. For example, an electronic spreadsheet is a program that allows a user to maintain a matrix, some of whose elements are data provided by the user. Other elements are computed based on the input data using potentially complex formulas. When a user changes one element, many other elements may have to be recomputed. By having a background thread do the recomputation, the interactive thread can allow the user to make additional changes while the computation is going on. Similarly, a third thread can handle periodic backups to disk on its own.

Now consider yet another example of where threads are useful: a server for a World Wide Web site. Requests for pages come in and the requested page is sent back to the client. At most Web sites, some pages are more commonly accessed than other pages. For example, Sony's home page is accessed far more than a page deep in the tree containing the technical specifications of some particular camcorder. Web servers use this fact to improve performance by maintaining a collection of heavily used pages in main memory to eliminate the need to go to disk to get them. Such a collection is called a **cache** and is used in many other contexts as well.

One way to organize the Web server is shown in Fig. 2-10(a). Here one thread, the **dispatcher**, reads incoming requests for work from the network. After examining the request, it chooses an idle (i.e., blocked) **worker thread** and hands it the request, possibly by writing a pointer to the message into a special word associated with each thread. The dispatcher then wakes up the sleeping worker, moving it from blocked state to ready state.

When the worker wakes up, it checks to see if the request can be satisfied from the Web page cache, to which all threads have access. If not, it starts a read operation to get the page from the disk and blocks until the disk operation completes. When the thread blocks on the disk operation, another thread is chosen to run, possibly the dispatcher, in order to acquire more work, or possibly another worker that is now ready to run.

This model allows the server to be written as a collection of sequential threads. The dispatcher's program consists of an infinite loop for getting a work

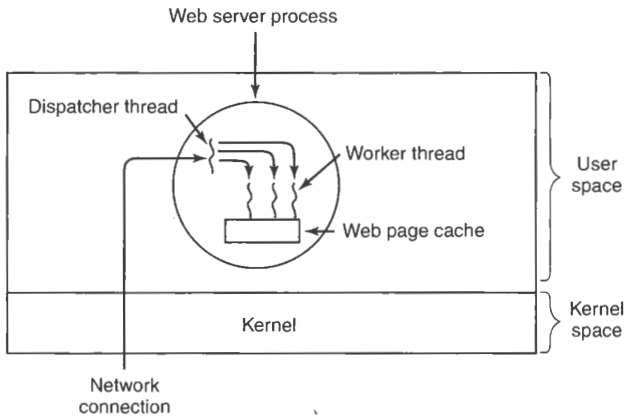


Figure 2-10. A multithreaded Web server.

request and handing it off to a worker. Each worker's code consists of an infinite loop consisting of accepting a request from the dispatcher and checking the Web cache to see if the page is present. If so, it is returned to the client and the worker blocks waiting for a new request. If not, it gets the page from the disk, returns it to the client, and blocks waiting for a new request.

A rough outline of the code is given in Fig. 2-11. Here, as in the rest of this book, *TRUE* is assumed to be the constant 1. Also, *buf* and *page* are structures appropriate for holding a work request and a Web page, respectively.

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

(a)

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

(b)

Figure 2-11. A rough outline of the code for Fig. 2-10. (a) Dispatcher thread. (b) Worker thread.

Consider how the Web server could be written in the absence of threads. One possibility is to have it operate as a single thread. The main loop of the Web server gets a request, examines it, and carries it out to completion before getting the next one. While waiting for the disk, the server is idle and does not process any other incoming requests. If the Web server is running on a dedicated

machine, as is commonly the case, the CPU is simply idle while the Web server is waiting for the disk. The net result is that many fewer requests/sec can be processed. Thus threads gain considerable performance, but each thread is programmed sequentially, in the usual way.

So far we have seen two possible designs: a multithreaded Web server and a single-threaded Web server. Suppose that threads are not available but the system designers find the performance loss due to single threading unacceptable. If a nonblocking version of the read system call is available, a third approach is possible. When a request comes in, the one and only thread examines it. If it can be satisfied from the cache, fine, but if not, a nonblocking disk operation is started.

The server records the state of the current request in a table and then goes and gets the next event. The next event may either be a request for new work or a reply from the disk about a previous operation. If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table and the reply processed. With nonblocking disk I/O, a reply probably will have to take the form of a signal or interrupt.

In this design, the "sequential process" model that we had in the first two cases is lost. The state of the computation must be explicitly saved and restored in the table every time the server switches from working on one request to another. In effect, we are simulating the threads and their stacks the hard way. A design like this in which each computation has a saved state and there exists some set of events that can occur to change the state is called a **finite-state machine**. This concept is widely used throughout computer science.

It should now be clear what threads have to offer. They make it possible to retain the idea of sequential processes that make blocking system calls (e.g., for disk I/O) and still achieve parallelism. Blocking system calls make programming easier and parallelism improves performance. The single-threaded server retains the ease of blocking system calls but gives up performance. The third approach achieves high performance through parallelism but uses nonblocking calls and interrupts and is thus is hard to program. These models are summarized in Fig. 2-12.

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Figure 2-12. Three ways to construct a server.

A third example where threads are useful is in applications that must process very large amounts of data. The normal approach is to read in a block of data, process it, and then write it out again. The problem here is that if only blocking system calls are available, the process blocks while data are coming in and data

are going out. Having the CPU go idle when there is lots of computing to do is clearly wasteful and should be avoided if possible.

Threads offer a solution. The process could be structured with an input thread, a processing thread, and an output thread. The input thread reads data into an input buffer. The processing thread takes data out of the input buffer, processes them, and puts the results in an output buffer. The output buffer writes these results back to disk. In this way, input, output, and processing can all be going on at the same time. Of course, this model only works if a system call blocks only the calling thread, not the entire process.

2.2.3 Implementing Threads in User Space

There are two main ways to implement a threads package: in user space and in the kernel. The choice is moderately controversial, and a hybrid implementation is also possible. We will now describe these methods, along with their advantages and disadvantages.

The first method is to put the threads package entirely in user space. The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes. The first, and most obvious, advantage is that a user-level threads package can be implemented on an operating system that does not support threads. All operating systems used to fall into this category, and even now some still do.

All of these implementations have the same general structure, which is illustrated in Fig. 2-13(a). The threads run on top of a run-time system, which is a collection of procedures that manage threads. We have seen four of these already: *thread_create*, *thread_exit*, *thread_wait*, and *thread_yield*, but usually there are more.

When threads are managed in user space, each process needs its own private **thread table** to keep track of the threads in that process. This table is analogous to the kernel's process table, except that it keeps track only of the per-thread properties such the each thread's program counter, stack pointer, registers, state, etc. The thread table is managed by the run-time system. When a thread is moved to ready state or blocked state, the information needed to restart it is stored in the thread table, exactly the same way as the kernel stores information about processes in the process table.

When a thread does something that may cause it to become blocked locally, for example, waiting for another thread in its process to complete some work, it calls a run-time system procedure. This procedure checks to see if the thread must be put into blocked state. If so, it stores the thread's registers (i.e., its own) in the thread table, looks in the table for a ready thread to run, and reloads the machine registers with the new thread's saved values. As soon as the stack pointer and program counter have been switched, the new thread comes to life

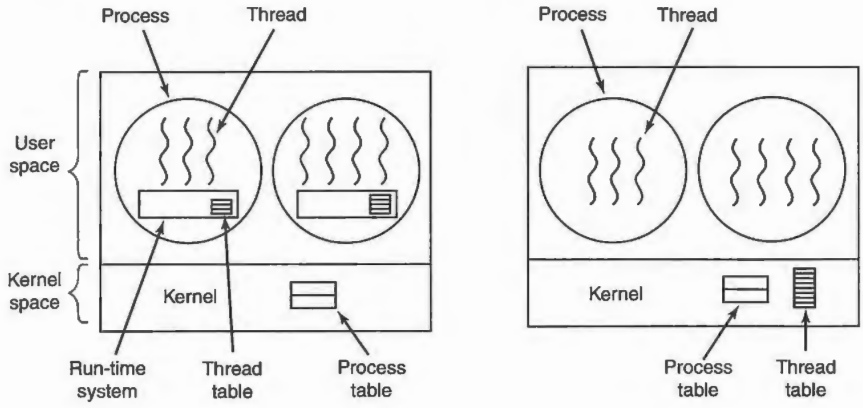


Figure 2-13. (a) A user-level threads package. (b) A threads package managed by the kernel.

again automatically. If the machine has an instruction to store all the registers and another one to load them all, the entire thread switch can be done in a handful of instructions. Doing thread switching like this is at least an order of magnitude faster than trapping to the kernel and is a strong argument in favor of user-level threads packages.

However, there is one key difference with processes. When a thread is finished running for the moment, for example, when it calls *thread_yield*, the code of *thread_yield* can save the thread's information in the thread table itself. Furthermore, it can then call the thread scheduler to pick another thread to run. The procedure that saves the thread's state and the scheduler are just local procedures, so invoking them is much more efficient than making a kernel call. Among other issues, no trap is needed, no context switch is needed, the memory cache need not be flushed, and so on. This makes thread scheduling very fast.

User-level threads also have other advantages. They allow each process to have its own customized scheduling algorithm. For some applications, for example, those with a garbage collector thread, not having to worry about a thread being stopped at an inconvenient moment is a plus. They also scale better, since kernel threads invariably require some table space and stack space in the kernel, which can be a problem if there are a very large number of threads.

Despite their better performance, user-level threads packages have some major problems. First among these is the problem of how blocking system calls are implemented. Suppose that a thread reads from the keyboard before any keys have been hit. Letting the thread actually make the system call is unacceptable, since this will stop all the threads. One of the main goals of having threads in the first place was to allow each one to use blocking calls, but to prevent one blocked

thread from affecting the others. With blocking system calls, it is hard to see how this goal can be achieved readily.

The system calls could all be changed to be nonblocking (e.g., a read on the keyboard would just return 0 bytes if no characters were already buffered), but requiring changes to the operating system is unattractive. Besides, one of the arguments for user-level threads was precisely that they could run with *existing* operating systems. In addition, changing the semantics of read will require changes to many user programs.

Another alternative is possible in the event that it is possible to tell in advance if a call will block. In some versions of UNIX, a system call, select, exists, which allows the caller to tell whether a prospective read will block. When this call is present, the library procedure *read* can be replaced with a new one that first does a select call and then only does the read call if it is safe (i.e., will not block). If the read call will block, the call is not made. Instead, another thread is run. The next time the run-time system gets control, it can check again to see if the read is now safe. This approach requires rewriting parts of the system call library, is inefficient and inelegant, but there is little choice. The code placed around the system call to do the checking is called a **jacket** or **wrapper**.

Somewhat analogous to the problem of blocking system calls is the problem of page faults. We will study these in Chap. 4. For the moment, it is sufficient to say that computers can be set up in such a way that not all of the program is in main memory at once. If the program calls or jumps to an instruction that is not in memory, a page fault occurs and the operating system will go and get the missing instruction (and its neighbors) from disk. This is called a page fault. The process is blocked while the necessary instruction is being located and read in. If a thread causes a page fault, the kernel, not even knowing about the existence of threads, naturally blocks the entire process until the disk I/O is complete, even though other threads might be runnable.

Another problem with user-level thread packages is that if a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU. Within a single process, there are no clock interrupts, making it impossible to schedule processes round-robin fashion (taking turns). Unless a thread enters the run-time system of its own free will, the scheduler will never get a chance.

One possible solution to the problem of threads running forever is to have the run-time system request a clock signal (interrupt) once a second to give it control, but this, too, is crude and messy to program. Periodic clock interrupts at a higher frequency are not always possible, and even if they are, the total overhead may be substantial. Furthermore, a thread might also need a clock interrupt, interfering with the run-time system's use of the clock.

Another, and probably the most devastating argument against user-level threads, is that programmers generally want threads precisely in applications where the threads block often, as, for example, in a multithreaded Web server.

These threads are constantly making system calls. Once a trap has occurred to the kernel to carry out the system call, it is hardly any more work for the kernel to switch threads if the old one has blocked, and having the kernel do this eliminates the need for constantly making select system calls that check to see if read system calls are safe. For applications that are essentially entirely CPU bound and rarely block, what is the point of having threads at all? No one would seriously propose computing the first *n* prime numbers or playing chess using threads because there is nothing to be gained by doing it that way.

2.2.4 Implementing Threads in the Kernel

Now let us consider having the kernel know about and manage the threads. No run-time system is needed in each, as shown in Fig. 2-13(b). Also, there is no thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system. When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction by updating the kernel thread table.

The kernel's thread table holds each thread's registers, state, and other information. The information is the same as with user-level threads, but it is now in the kernel instead of in user space (inside the run-time system). This information is a subset of the information that traditional kernels maintain about each of their single-threaded processes, that is, the process state. In addition, the kernel also maintains the traditional process table to keep track of processes.

All calls that might block a thread are implemented as system calls, at considerably greater cost than a call to a run-time system procedure. When a thread blocks, the kernel, at its option, can run either another thread from the same process (if one is ready), or a thread from a different process. With user-level threads, the run-time system keeps running threads from its own process until the kernel takes the CPU away from it (or there are no ready threads left to run).

Due to the relatively greater cost of creating and destroying threads in the kernel, some systems take an environmentally correct approach and recycle their threads. When a thread is destroyed, it is marked as not runnable, but its kernel data structures are not otherwise affected. Later, when a new thread must be created, an old thread is reactivated, saving some overhead. Thread recycling is also possible for user-level threads, but since the thread management overhead is much smaller, there is less incentive to do this.

Kernel threads do not require any new, nonblocking system calls. In addition, if one thread in a process causes a page fault, the kernel can easily check to see if the process has any other runnable threads, and if so, run one of them while waiting for the required page to be brought in from the disk. Their main disadvantage is that the cost of a system call is substantial, so if thread operations (creation, termination, etc.) are common, much more overhead will be incurred.

2.2.5 Hybrid Implementations

Various ways have been investigated to try to combine the advantages of user-level threads with kernel-level threads. One way is use kernel-level threads and then multiplex user-level threads onto some or all of the kernel threads, as shown in Fig. 2-14.

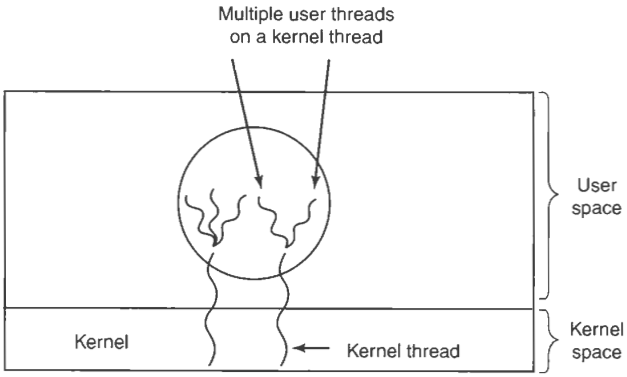


Figure 2-14. Multiplexing user-level threads onto kernel-level threads.

In this design, the kernel is aware of only the kernel-level threads and schedules those. Some of those threads may have multiple user-level threads multiplexed on top of them. These user-level threads are created, destroyed, and scheduled just like user-level threads in a process that runs on an operating system without multithreading capability. In this model, each kernel-level thread has some set of user-level threads that take turns using it.

2.2.6 Scheduler Activations

Various researchers have attempted to combine the advantage of user threads (good performance) with the advantage of kernel threads (not having to use a lot of tricks to make things work). Below we will describe one such approach devised by Anderson et al. (1992), called **scheduler activations**. Related work is discussed by Edler et al. (1988) and Scott et al. (1990).

The goals of the scheduler activation work are to mimic the functionality of kernel threads, but with the better performance and greater flexibility usually associated with threads packages implemented in user space. In particular, user threads should not have to make special nonblocking system calls or check in advance if it is safe to make certain system calls. Nevertheless, when a thread blocks on a system call or on a page fault, it should be possible to run other threads within the same process, if any are ready.

Efficiency is achieved by avoiding unnecessary transitions between user and kernel space. If a thread blocks waiting for another thread to do something, for example, there is no reason to involve the kernel, thus saving the overhead of the kernel-user transition. The user-space run-time system can block the synchronizing thread and schedule a new one by itself.

When scheduler activations are used, the kernel assigns a certain number of virtual processors to each process and lets the (user-space) run-time system allocate threads to processors. This mechanism can also be used on a multiprocessor where the virtual processors may be real CPUs. The number of virtual processors allocated to a process is initially one, but the process can ask for more and can also return processors it no longer needs. The kernel can also take back virtual processors already allocated in order to assign them to other, more needy, processes.

The basic idea that makes this scheme work is that when the kernel knows that a thread has blocked (e.g., by its having executed a blocking system call or caused a page fault), the kernel notifies the process' run-time system, passing as parameters on the stack the number of the thread in question and a description of the event that occurred. The notification happens by having the kernel activate the run-time system at a known starting address, roughly analogous to a signal in UNIX. This mechanism is called an **upcall**.

Once activated like this, the run-time system can reschedule its threads, typically by marking the current thread as blocked and taking another thread from the ready list, setting up its registers, and restarting it. Later, when the kernel learns that the original thread can run again (e.g., the pipe it was trying to read from now contains data, or the page it faulted over has been brought in from disk), it makes another upcall to the run-time system to inform it of this event. The run-time system, at its own discretion, can either restart the blocked thread immediately, or put it on the ready list to be run later.

When a hardware interrupt occurs while a user thread is running, the interrupted CPU switches into kernel mode. If the interrupt is caused by an event not of interest to the interrupted process, such as completion of another process' I/O, when the interrupt handler has finished, it puts the interrupted thread back in the state it was in before the interrupt. If, however, the process is interested in the interrupt, such as the arrival of a page needed by one of the process' threads, the interrupted thread is not restarted. Instead, the interrupted thread is suspended and the run-time system started on that virtual CPU, with the state of the interrupted thread on the stack. It is then up to the run-time system to decide which thread to schedule on that CPU: the interrupted one, the newly ready one, or some third choice.

An objection to scheduler activations is the fundamental reliance on upcalls, a concept that violates the structure inherent in any layered system. Normally, layer n offers certain services that layer $n + 1$ can call on, but layer n may not call procedures in layer $n + 1$. Upcalls do not follow this fundamental principle.

2.2.7 Pop-Up Threads

Threads are frequently useful in distributed systems. An important example is how incoming messages, for example requests for service, are handled. The traditional approach is to have a process or thread that is blocked on a receive system call waiting for an incoming message. When a message arrives, it accepts the message and processes it.

However, a completely different approach is also possible, in which the arrival of a message causes the system to create a new thread to handle the message. Such a thread is called a **pop-up thread** and is illustrated in Fig. 2-15. A key advantage of pop-up threads is that since they are brand new, they do not have any history—registers, stack, etc. that must be restored. Each one starts out fresh and each one is identical to all the others. This makes it possible to create such a thread quickly. The new thread is given the incoming message to process. The result of using pop-up threads is that the latency between message arrival and the start of processing can be made very short.

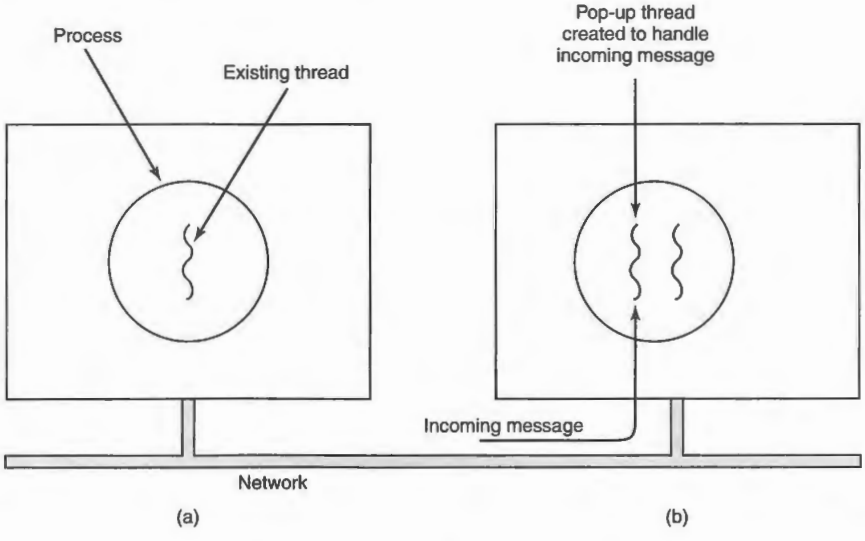


Figure 2-15. Creation of a new thread when a message arrives. (a) Before the message arrives. (b) After the message arrives.

Some advance planning is needed when pop-up threads are used. For example, in which process does the thread run? If the system supports threads running in the kernel's context, the thread may run there (which is why we have not shown the kernel in Fig. 2-15). Having the pop-up thread run in kernel space is usually easier and faster than putting it in user space. Also, a pop-up thread in kernel

space can easily access all the kernel's tables and the I/O devices, which may be needed for interrupt processing. On the other hand, a buggy kernel thread can do more damage than a buggy user thread. For example, if it runs too long and there is no way to preempt it, incoming data may be lost.

2.2.8 Making Single-Threaded Code Multithreaded

Many existing programs were written for single-threaded processes. Converting these to multithreading is much trickier than it may at first appear. Below we will examine just a few of the pitfalls.

As a start, the code of a thread normally consists of multiple procedures, just like a process. These may have local variables, global variables, and procedure parameters. Local variables and parameters do not cause any trouble, but variables that are global to a thread but not global to the entire program do. These are variables that are global in the sense that many procedures within the thread use them (as they might use any global variable), but other threads should logically leave them alone.

As an example, consider the *errno* variable maintained by UNIX. When a process (or a thread) makes a system call that fails, the error code is put into *errno*. In Fig. 2-16, thread 1 executes the system call access to find out if it has permission to access a certain file. The operating system returns the answer in the global variable *errno*. After control has returned to thread 1, but before it has a chance to read *errno*, the scheduler decides that thread 1 has had enough CPU time for the moment and decides to switch to thread 2. Thread 2 executes an open call that fails, which causes *errno* to be overwritten and thread 1's access code to be lost forever. When thread 1 starts up later, it will read the wrong value and behave incorrectly.

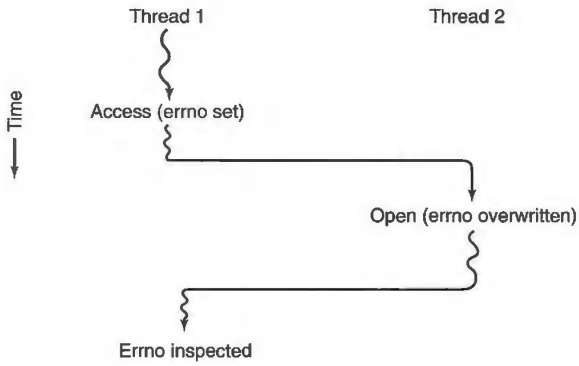


Figure 2-16. Conflicts between threads over the use of a global variable.

Various solutions to this problem are possible. One is to prohibit global variables altogether. However worthy this ideal may be, it conflicts with much existing software. Another is to assign each thread its own private global variables, as shown in Fig. 2-17. In this way, each thread has its own private copy of *errno* and other global variables, so conflicts are avoided. In effect, this decision creates a new scoping level, variables visible to all the procedures of a thread, in addition to the existing scoping levels of variables visible only to one procedure and variables visible everywhere in the program.

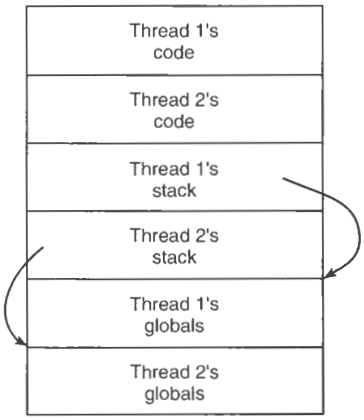


Figure 2-17. Threads can have private global variables.

Accessing the private global variables is a bit tricky, however, since most programming languages have a way of expressing local variables and global variables, but not intermediate forms. It is possible to allocate a chunk of memory for the globals and pass it to each procedure in the thread, as an extra parameter. While hardly an elegant solution, it works.

Alternatively, new library procedures can be introduced to create, set, and read these thread-wide global variables. The first call might look like this:

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

It allocates storage for a pointer called *bufptr* on the heap or in a special storage area reserved for the calling thread. No matter where the storage is allocated, only the calling thread has access to the global variable. If another thread creates a global variable with the same name, it gets a different storage location that does not conflict with the existing one.

Two calls are needed to access global variables: one for writing them and the other for reading them. For writing, something like

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

will do. It stores the value of a pointer in the storage location previously created by the call to *create_global*. To read a global variable, the call might look like

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

It returns the address stored in the global variable, so its data can be accessed.

The next problem turning a single-threaded program into a multithreaded program is that many library procedures are not reentrant. That is, they were not designed to have a second call made to any given procedure while a previous call has not yet finished. For example, sending a message over the network may well be programmed to assemble the message in a fixed buffer within the library, then to trap to the kernel to send it. What happens if one thread has assembled its message in the buffer, then a clock interrupt forces a switch to a second thread that immediately overwrites the buffer with its own message?

Similarly, memory allocation procedures, such as *malloc* in UNIX, maintain crucial tables about memory usage, for example, a linked list of available chunks of memory. While *malloc* is busy updating these lists, they may temporarily be in an inconsistent state, with pointers that point nowhere. If a thread switch occurs while the tables are inconsistent and a new call comes in from a different thread, an invalid pointer may be used, leading to a program crash. Fixing all these problems properly effectively means rewriting the entire library.

A different solution is to provide each procedure with a jacket that sets a bit to mark the library as in use. Any attempt for another thread to use a library procedure while a previous call has not yet completed is blocked. Although this approach can be made to work, it greatly eliminates potential parallelism.

Next, consider signals. Some signals are logically thread specific, whereas others are not. For example, if a thread calls alarm, it makes sense for the resulting signal to go to the thread that made the call. However, when threads are implemented entirely in user space, the kernel does not even know about threads and can hardly direct the signal to the right one. An additional complication occurs if a process may only have one alarm at a time pending and several threads call alarm independently.

Other signals, such as keyboard interrupt, are not thread specific. Who should catch them? One designated thread? All the threads? A newly created pop-up thread? Furthermore, what happens if one thread changes the signal handlers without telling other threads? And what happens if one thread wants to catch a particular signal (say, the user hitting CTRL-C), and another thread wants this signal to terminate the process? This situation can arise if one or more threads run standard library procedures and others are user-written. Clearly, these wishes are incompatible. In general, signals are difficult enough to manage in a single-threaded environment. Going to a multithreaded environment does not make them any easier to handle.

One last problem introduced by threads is stack management. In many systems, when a process' stack overflows, the kernel just provides that process with

more stack automatically. When a process has multiple threads, it must also have multiple stacks. If the kernel is not aware of all these stacks, it cannot grow them automatically upon stack fault. In fact, it may not even realize that a memory fault is related to stack growth.

These problems are certainly not insurmountable, but they do show that just introducing threads into an existing system without a fairly substantial system redesign is not going to work at all. The semantics of system calls may have to be redefined and libraries have to be rewritten, at the very least. And all of these things must be done in such a way as to remain backward compatible with existing programs for the limiting case of a process with only one thread. For additional information about threads, see (Hauser et al., 1993; and Marsh et al., 1991).

2.3 INTERPROCESS COMMUNICATION

Processes frequently need to communicate with other processes. For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on down the line. Thus there is a need for communication between processes, preferably in a well-structured way not using interrupts. In the following sections we will look at some of the issues related to this **InterProcess Communication** or **IPC**.

Very briefly, there are three issues here. The first was alluded to above: how one process can pass information to another. The second has to do with making sure two or more processes do not get into each other's way when engaging in critical activities (suppose two processes each try to grab the last 1 MB of memory). The third concerns proper sequencing when dependencies are present: if process *A* produces data and process *B* prints them, *B* has to wait until *A* has produced some data before starting to print. We will examine all three of these issues starting in the next section.

It is also important to mention that two of these issues apply equally well to threads. The first one—passing information—is easy for threads since they share a common address space (threads in different address spaces that need to communicate fall under the heading of communicating processes). However, the other two—keeping out of each other's hair and proper sequencing—apply equally well to threads. The same problems exist and the same solutions apply. Below we will discuss the problem in the context of processes, but please keep in mind that the same problems and solutions also apply to threads.

2.3.1 Race Conditions

In some operating systems, processes that are working together may share some common storage that each one can read and write. The shared storage may be in main memory (possibly in a kernel data structure) or it may be a shared file;

the location of the shared memory does not change the nature of the communication or the problems that arise. To see how interprocess communication works in practice, let us consider a simple but common example: a print spooler. When a process wants to print a file, it enters the file name in a special **spooler directory**. Another process, the **printer daemon**, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory.

Imagine that our spooler directory has a very large number of slots, numbered 0, 1, 2, ..., each one capable of holding a file name. Also imagine that there are two shared variables, *out*, which points to the next file to be printed, and *in*, which points to the next free slot in the directory. These two variables might well be kept on a two-word file available to all processes. At a certain instant, slots 0 to 3 are empty (the files have already been printed) and slots 4 to 6 are full (with the names of files queued for printing). More or less simultaneously, processes *A* and *B* decide they want to queue a file for printing. This situation is shown in Fig. 2-18.

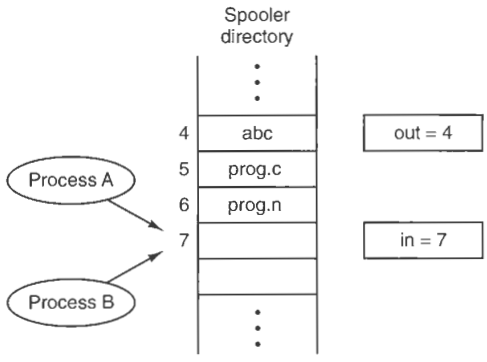


Figure 2-18. Two processes want to access shared memory at the same time.

In jurisdictions where Murphy's law† is applicable, the following might happen. Process *A* reads *in* and stores the value, 7, in a local variable called *next_free_slot*. Just then a clock interrupt occurs and the CPU decides that process *A* has run long enough, so it switches to process *B*. Process *B* also reads *in*, and also gets a 7. It too stores it in *its* local variable *next_free_slot*. At this instant both processes think that the next available slot is 7.

Process *B* now continues to run. It stores the name of its file in slot 7 and updates *in* to be an 8. Then it goes off and does other things.

Eventually, process *A* runs again, starting from the place it left off. It looks at *next_free_slot*, finds a 7 there, and writes its file name in slot 7, erasing the name

† If something can go wrong, it will.

that process *B* just put there. Then it computes *next_free_slot* + 1, which is 8, and sets *in* to 8. The spooler directory is now internally consistent, so the printer daemon will not notice anything wrong, but process *B* will never receive any output. User *B* will hang around the printer room for years, wistfully hoping for output that never comes. Situations like this, where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called **race conditions**. Debugging programs containing race conditions is no fun at all. The results of most test runs are fine, but once in a rare while something weird and unexplained happens.

2.3.2 Critical Regions

How do we avoid race conditions? The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else is to find some way to prohibit more than one process from reading and writing the shared data at the same time. Put in other words, what we need is **mutual exclusion**, that is, some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing. The difficulty above occurred because process *B* started using one of the shared variables before process *A* was finished with it. The choice of appropriate primitive operations for achieving mutual exclusion is a major design issue in any operating system, and a subject that we will examine in great detail in the following sections.

The problem of avoiding race conditions can also be formulated in an abstract way. Part of the time, a process is busy doing internal computations and other things that do not lead to race conditions. However, sometimes a process have to access shared memory or files, or doing other critical things that can lead to races. That part of the program where the shared memory is accessed is called the **critical region** or **critical section**. If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races.

Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data. We need four conditions to hold to have a good solution:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

In an abstract sense, the behavior that we want is shown in Fig. 2-19. Here process *A* enters its critical region at time T_1. A little later, at time T_2 process *B* attempts to enter its critical region but fails because another process is already in

its critical region and we allow only one at a time. Consequently, B is temporarily suspended until time T_3 when A leaves its critical region, allowing B to enter immediately. Eventually B leaves (at T_4) and we are back to the original situation with no processes in their critical regions.

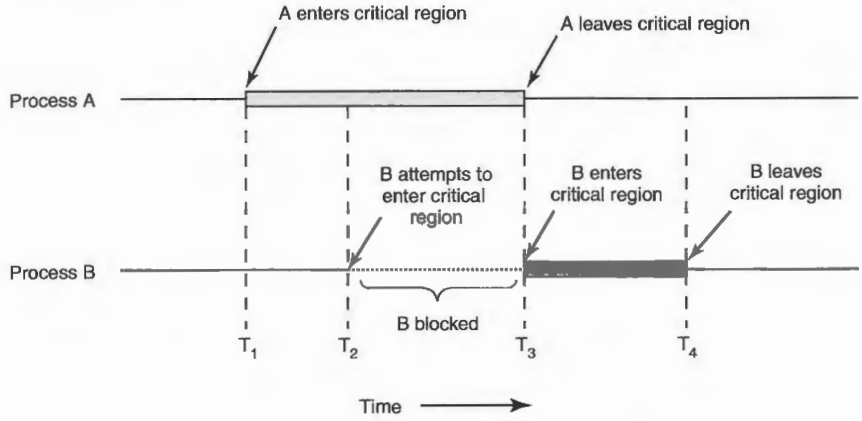


Figure 2-19. Mutual exclusion using critical regions.

2.3.3 Mutual Exclusion with Busy Waiting

In this section we will examine various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its critical region, no other process will enter *its* critical region and cause trouble.

Disabling Interrupts

The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did it, and never turned them on again? That could be the end of the system. Furthermore, if the system is a multiprocessor, with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

On the other hand, it is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists. If an interrupt occurred while the list of ready processes, for example, was in an inconsistent state, race conditions could occur. The conclusion is: disabling interrupts is often a useful technique within the operating system itself but is not appropriate as a general mutual exclusion mechanism for user processes.

Lock Variables

As a second attempt, let us look for a software solution. Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

Now you might think that we could get around this problem by first reading out the lock value, then checking it again just before storing into it, but that really does not help. The race now occurs if the second process modifies the lock just after the first process has finished its second check.

Strict Alternation

A third approach to the mutual exclusion problem is shown in Fig. 2-20. This program fragment, like nearly all the others in this book, is written in C. C was chosen here because real operating systems are virtually always written in C (or occasionally C++), but hardly ever in languages like Java, Modula 3, or Pascal. C is powerful, efficient, and predictable, characteristics critical for writing operating systems. Java, for example, is not predictable because it might run out of storage at a critical moment and need to invoke the garbage collector at a most inopportune time. This cannot happen in C because there is no garbage collection in C. A quantitative comparison of C, C++, Java, and four other languages is given in (Prechelt, 2000).

In Fig. 2-20, the integer variable *turn*, initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects *turn*, finds it to be 0, and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing *turn* to see when it becomes 1. Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time.

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

(a)

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

(b)

Figure 2-20. A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a **spin lock**.

When process 0 leaves the critical region, it sets *turn* to 1, to allow process 1 to enter its critical region. Suppose that process 1 finishes its critical region quickly, so both processes are in their noncritical regions, with *turn* set to 0. Now process 0 executes its whole loop quickly, exiting its critical region and setting *turn* to 1. At this point *turn* is 1 and both processes are executing in their noncritical regions.

Suddenly, process 0 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now, because *turn* is 1 and process 1 is busy with its noncritical region. It hangs in its while loop until process 1 sets *turn* to 0. Put differently, taking turns is not a good idea when one of the processes is much slower than the other.

This situation violates condition 3 set out above: process 0 is being blocked by a process not in its critical region. Going back to the spooler directory discussed above, if we now associate the critical region with reading and writing the spooler directory, process 0 would not be allowed to print another file because process 1 was doing something else.

In fact, this solution requires that the two processes strictly alternate in entering their critical regions, for example, in spooling files. Neither one would be permitted to spool two in a row. While this algorithm does avoid all races, it is not really a serious candidate as a solution because it violates condition 3.

Peterson's Solution

By combining the idea of taking turns with the idea of lock variables and warning variables, a Dutch mathematician, T. Dekker, was the first one to devise a software solution to the mutual exclusion problem that does not require strict alternation. For a discussion of Dekker's algorithm, see (Dijkstra, 1965).

In 1981, G.L. Peterson discovered a much simpler way to achieve mutual exclusion, thus rendering Dekker's solution obsolete. Peterson's algorithm is shown in Fig. 2-21. This algorithm consists of two procedures written in ANSI C,

which means that function prototypes should be supplied for all the functions defined and used. However, to save space, we will not show the prototypes in this or subsequent examples.

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Figure 2-21. Peterson's solution for achieving mutual exclusion.

Before using the shared variables (i.e., before entering its critical region), each process calls *enter_region* with its own process number, 0 or 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls *leave_region* to indicate that it is done and to allow the other process to enter, if it so desires.

Let us see how this solution works. Initially neither process is in its critical region. Now process 0 calls *enter_region*. It indicates its interest by setting its array element and sets *turn* to 0. Since process 1 is not interested, *enter_region* returns immediately. If process 1 now calls *enter_region*, it will hang there until *interested*[0] goes to *FALSE*, an event that only happens when process 0 calls *leave_region* to exit the critical region.

Now consider the case that both processes call *enter_region* almost simultaneously. Both will store their process number in *turn*. Whichever store is done last is the one that counts; the first one is overwritten and lost. Suppose that process 1 stores last, so *turn* is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region until process 0 exits its critical region.

The TSL Instruction

Now let us look at a proposal that requires a little help from the hardware. Many computers, especially those designed with multiple processors in mind, have an instruction

```
TSL RX,LOCK
```

(Test and Set Lock) that works as follows. It reads the contents of the memory word *lock* into register RX and then stores a nonzero value at the memory address *lock*. The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

To use the TSL instruction, we will use a shared variable, *lock*, to coordinate access to shared memory. When *lock* is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets *lock* back to 0 using an ordinary move instruction.

How can this instruction be used to prevent two processes from simultaneously entering their critical regions? The solution is given in Fig. 2-22. There a four-instruction subroutine in a fictitious (but typical) assembly language is shown. The first instruction copies the old value of *lock* to the register and then sets *lock* to 1. Then the old value is compared with 0. If it is nonzero, the lock was already set, so the program just goes back to the beginning and tests it again. Sooner or later it will become 0 (when the process currently in its critical region is done with its critical region), and the subroutine returns, with the lock set. Clearing the lock is simple. The program just stores a 0 in *lock*. No special instructions are needed.

```
enter_region:
    TSL REGISTER,LOCK          | copy lock to register and set lock to 1
    CMP REGISTER,#0            | was lock zero?
    JNE enter_region           | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0               | store a 0 in lock
    RET | return to caller
```

Figure 2-22. Entering and leaving a critical region using the TSL instruction.

One solution to the critical region problem is now straightforward. Before entering its critical region, a process calls *enter_region*, which does busy waiting until the lock is free; then it acquires the lock and returns. After the critical region

the process calls *leave_region*, which stores a 0 in *lock*. As with all solutions based on critical regions, the processes must call *enter_region* and *leave_region* at the correct times for the method to work. If a process cheats, the mutual exclusion will fail.

2.3.4 Sleep and Wakeup

Both Peterson's solution and the solution using TSL are correct, but both have the defect of requiring busy waiting. In essence, what these solutions do is this: when a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not, the process just sits in a tight loop waiting until it is.

Not only does this approach waste CPU time, but it can also have unexpected effects. Consider a computer with two processes, *H*, with high priority and *L*, with low priority. The scheduling rules are such that *H* is run whenever it is in ready state. At a certain moment, with *L* in its critical region, *H* becomes ready to run (e.g., an I/O operation completes). *H* now begins busy waiting, but since *L* is never scheduled while *H* is running, *L* never gets the chance to leave its critical region, so *H* loops forever. This situation is sometimes referred to as the **priority inversion problem**.

Now let us look at some interprocess communication primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions. One of the simplest is the pair sleep and wakeup. Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened. Alternatively, both sleep and wakeup each have one parameter, a memory address used to match up sleeps with wakeups.

The Producer-Consumer Problem

As an example of how these primitives can be used, let us consider the **producer-consumer** problem (also known as the **bounded-buffer** problem). Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out. (It is also possible to generalize the problem to have *m* producers and *n* consumers, but we will only consider the case of one producer and one consumer because this assumption simplifies the solutions).

Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

This approach sounds simple enough, but it leads to the same kinds of race conditions we saw earlier with the spooler directory. To keep track of the number

of items in the buffer, we will need a variable, *count*. If the maximum number of items the buffer can hold is *N*, the producer's code will first test to see if *count* is *N*. If it is, the producer will go to sleep; if it is not, the producer will add an item and increment *count*.

The consumer's code is similar: first test *count* to see if it is 0. If it is, go to sleep; if it is nonzero, remove an item and decrement the counter. Each of the processes also tests to see if the other should be awakened, and if so, wakes it up. The code for both producer and consumer is shown in Fig. 2-23.

```
#define N 100                                /* number of slots in the buffer */
int count = 0;                               /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                          /* repeat forever */
          item = produce_item( );            /* generate next item */
          if (count == N) sleep( );          /* if buffer is full, go to sleep */
          insert_item(item);                 /* put item in buffer */
          count = count + 1;                 /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);  /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                          /* repeat forever */
          if (count == 0) sleep( );          /* if buffer is empty, got to sleep */
          item = remove_item( );             /* take item out of buffer */
          count = count − 1;                 /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                /* print item */
     }
}
```

Figure 2-23. The producer-consumer problem with a fatal race condition.

To express system calls such as sleep and wakeup in C, we will show them as calls to library routines. They are not part of the standard C library but presumably would be available on any system that actually had these system calls. The procedures *insert_item* and *remove_item*, which are not shown, handle the bookkeeping of putting items into the buffer and taking items out of the buffer.

Now let us get back to the race condition. It can occur because access to *count* is unconstrained. The following situation could possibly occur. The buffer is empty and the consumer has just read *count* to see if it is 0. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. The producer inserts an item in the buffer, increments *count*, and notices that it is now 1. Reasoning that *count* was just 0, and thus the consumer must be sleeping, the producer calls *wakeup* to wake the consumer up.

Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost. When the consumer next runs, it will test the value of *count* it previously read, find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. If it were not lost, everything would work. A quick fix is to modify the rules to add a **wakeup waiting bit** to the picture. When a wakeup is sent to a process that is still awake, this bit is set. Later, when the process tries to go to sleep, if the wakeup waiting bit is on, it will be turned off, but the process will stay awake. The wakeup waiting bit is a piggy bank for wakeup signals.

While the wakeup waiting bit saves the day in this simple example, it is easy to construct examples with three or more processes in which one wakeup waiting bit is insufficient. We could make another patch and add a second wakeup waiting bit, or maybe 8 or 32 of them, but in principle the problem is still there.

2.3.5 Semaphores

This was the situation in 1965, when E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use. In his proposal, a new variable type, called a **semaphore**, was introduced. A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.

Dijkstra proposed having two operations, down and up (generalizations of sleep and wakeup, respectively). The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues. If the value is 0, the process is put to sleep without completing the down for the moment. Checking the value, changing it, and possibly going to sleep, is all done as a single, indivisible **atomic action**. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions.

The up operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its down. Thus, after an up on a semaphore with processes

sleeping on it, the semaphore will still be 0, but there will be one fewer process sleeping on it. The operation of incrementing the semaphore and waking up one process is also indivisible. No process ever blocks doing an up, just as no process ever blocks doing a wakeup in the earlier model.

As an aside, in Dijkstra's original paper, he used the names P and V instead of down and up, respectively, but since these have no mnemonic significance to people who do not speak Dutch (and only marginal significance to those who do), we will use the terms down and up instead. These were first introduced in Algol 68.

Solving the Producer-Consumer Problem using Semaphores

Semaphores solve the lost-wakeup problem, as shown in Fig. 2-24. It is essential that they be implemented in an indivisible way. The normal way is to implement up and down as system calls, with the operating system briefly disabling all interrupts while it is testing the semaphore, updating it, and putting the process to sleep, if necessary. As all of these actions take only a few instructions, no harm is done in disabling interrupts. If multiple CPUs are being used, each semaphore should be protected by a lock variable, with the TSL instruction used to make sure that only one CPU at a time examines the semaphore. Be sure you understand that using TSL to prevent several CPUs from accessing the semaphore at the same time is quite different from busy waiting by the producer or consumer waiting for the other to empty or fill the buffer. The semaphore operation will only take a few microseconds, whereas the producer or consumer might take arbitrarily long.

This solution uses three semaphores: one called *full* for counting the number of slots that are full, one called *empty* for counting the number of slots that are empty, and one called *mutex* to make sure the producer and consumer do not access the buffer at the same time. *Full* is initially 0, *empty* is initially equal to the number of slots in the buffer, and *mutex* is initially 1. Semaphores that are initialized to 1 and used by two or more processes to ensure that only one of them can enter its critical region at the same time are called **binary semaphores**. If each process does a down just before entering its critical region and an up just after leaving it, mutual exclusion is guaranteed.

Now that we have a good interprocess communication primitive at our disposal, let us go back and look at the interrupt sequence of Fig. 2-5 again. In a system using semaphores, the natural way to hide interrupts is to have a semaphore, initially set to 0, associated with each I/O device. Just after starting an I/O device, the managing process does a down on the associated semaphore, thus blocking immediately. When the interrupt comes in, the interrupt handler then does an up on the associated semaphore, which makes the relevant process ready to run again. In this model, step 5 in Fig. 2-5 consists of doing an up on the device's semaphore, so that in step 6 the scheduler will be able to run the device manager. Of course, if several processes are now ready, the scheduler may choose to run an

```
#define N 100                          /* number of slots in the buffer */
typedef int semaphore;                 /* semaphores are a special kind of int */
semaphore mutex = 1;                   /* controls access to critical region */
semaphore empty = N;                   /* counts empty buffer slots */
semaphore full = 0;                    /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                     /* TRUE is the constant 1 */
        item = produce_item( );        /* generate something to put in buffer */
        down(&empty);                  /* decrement empty count */
        down(&mutex);                  /* enter critical region */
        insert_item(item);             /* put new item in buffer */
        up(&mutex);                    /* leave critical region */
        up(&full);                     /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                     /* infinite loop */
        down(&full);                   /* decrement full count */
        down(&mutex);                  /* enter critical region */
        item = remove_item( );         /* take item from buffer */
        up(&mutex);                    /* leave critical region */
        up(&empty);                    /* increment count of empty slots */
        consume_item(item);            /* do something with the item */
    }
}
```

Figure 2-24. The producer-consumer problem using semaphores.

even more important process next. We will look at some of the algorithms used for scheduling later on in this chapter.

In the example of Fig. 2-24, we have actually used semaphores in two different ways. This difference is important enough to make explicit. The *mutex* semaphore is used for mutual exclusion. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables. This mutual exclusion is required to prevent chaos. We will study mutual exclusion and how to achieve it more in the next section.

The other use of semaphores is for **synchronization**. The *full* and *empty* semaphores are needed to guarantee that certain event sequences do or do not occur. In this case, they ensure that the producer stops running when the buffer is full, and the consumer stops running when it is empty. This use is different from mutual exclusion.

2.3.6 Mutexes

When the semaphore's ability to count is not needed, a simplified version of the semaphore, called a mutex, is sometimes used. Mutexes are good only for managing mutual exclusion to some shared resource or piece of code. They are easy and efficient to implement, which makes them especially useful in thread packages that are implemented entirely in user space.

A **mutex** is a variable that can be in one of two states: unlocked or locked. Consequently, only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked. Two procedures are used with mutexes. When a thread (or process) needs access to a critical region, it calls *mutex_lock*. If the mutex is current unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.

On the other hand, if the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls *mutex_unlock*. If multiple threads are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.

Because mutexes are so simple, they can easily be implemented in user space if a TSL instruction is available. The code for *mutex_lock* and *mutex_unlock* for use with a user-level threads package are shown in Fig. 2-25.

```
mutex_lock:
     TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
     CMP REGISTER,#0             | was mutex zero?
     JZE ok                      | if it was zero, mutex was unlocked, so return
     CALL thread_yield           | mutex is busy; schedule another thread
     JMP mutex_lock              | try again later
ok:  RET | return to caller; critical region entered


mutex_unlock:
     MOVE MUTEX,#0               | store a 0 in mutex
     RET | return to caller
```

Figure 2-25. Implementation of *mutex_lock* and *mutex_unlock*.

The code of *mutex_lock* is similar to the code of *enter_region* of Fig. 2-22 but with a crucial difference. When *enter_region* fails to enter the critical region, it keeps testing the lock repeatedly (busy waiting). Eventually, the clock runs out

and some other process is scheduled to run. Sooner or later the process holding the lock gets to run and releases it.

With threads, the situation is different because there is no clock that stops threads that have run too long. Consequently, a thread that tries to acquire a lock by busy waiting will loop forever and never acquire the lock because it never allows any other thread to run and release the lock.

That is where the difference between *enter_region* and *mutex_lock* comes in. When the later fails to acquire a lock, it calls *thread_yield* to give up the CPU to another thread. Consequently there is no busy waiting. When the thread runs the next time, it tests the lock again.

Since *thread_yield* is just a call to the thread scheduler in user space, it is very fast. As a consequence, neither *mutex_lock* nor *mutex_unlock* requires any kernel calls. Using them, user-level threads can synchronize entirely in user space using procedures that require only a handful of instructions.

The mutex system that we have described above is a bare bones set of calls. With all software, there is always a demand for more features, and synchronization primitives are no exception. For example, sometimes a thread package offers a call *mutex_trylock* that either acquires the lock or returns a code for failure, but does not block. This call gives the thread the flexibility to decide what to do next if there are alternatives to just waiting.

Up until now there is an issue that we have glossed over lightly but which is worth at least making explicit. With a user-space threads package there is no problem with multiple threads having access to the same mutex since all the threads operate in a common address space. However, with most of the earlier solutions, such as Peterson's algorithm and semaphores, there is an unspoken assumption that multiple processes have access to at least some shared memory, perhaps only one word, but something. If processes have disjoint address spaces, as we have consistently said, how can they share the *turn* variable in Peterson's algorithm, or semaphores or a common buffer?

There are two answers. First, some of the shared data structures, such as the semaphores, can be stored in the kernel and only accessed via system calls. This approach eliminates the problem. Second, most modern operating systems (including UNIX and Windows) offer a way for processes to share some portion of their address space with other processes. In this way, buffers and other data structures can be shared. In the worst case, that nothing else is possible, a shared file can be used.

If two or more processes share most or all of their address spaces, the distinction between processes and threads becomes somewhat blurred but is nevertheless present. Two processes that share a common address space still have different open files, alarm timers, and other per-process properties, whereas the threads within a single process share them. And it is always true that multiple processes sharing a common address space never have the efficiency of user-level threads since the kernel is deeply involved in their management.

2.3.7 Monitors

With semaphores interprocess communication looks easy, right? Forget it. Look closely at the order of the downs before inserting or removing items from the buffer in Fig. 2-24. Suppose that the two downs in the producer's code were reversed in order, so *mutex* was decremented before *empty* instead of after it. If the buffer were completely full, the producer would block, with *mutex* set to 0. Consequently, the next time the consumer tried to access the buffer, it would do a down on *mutex*, now 0, and block too. Both processes would stay blocked forever and no more work would ever be done. This unfortunate situation is called a deadlock. We will study deadlocks in detail in Chap. 3.

This problem is pointed out to show how careful you must be when using semaphores. One subtle error and everything comes to a grinding halt. It is like programming in assembly language, only worse, because the errors are race conditions, deadlocks, and other forms of unpredictable and irreproducible behavior.

To make it easier to write correct programs, Hoare (1974) and Brinch Hansen (1975) proposed a higher-level synchronization primitive called a **monitor**. Their proposals differed slightly, as described below. A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package. Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor. Figure 2-26 illustrates a monitor written in an imaginary language, Pidgin Pascal.

```
monitor example
    integer i;
    condition c;

    procedure producer( );
     .
     .
     .
     end;

     procedure consumer( );
     .    .    .
     end;
end monitor;
```

Figure 2-26. A monitor.

Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant. Monitors are a programming language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls. Typically, when a process calls a monitor procedure, the first few

instructions of the procedure will check to see if any other process is currently active within the monitor. If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

It is up to the compiler to implement the mutual exclusion on monitor entries, but a common way is to use a mutex or binary semaphore. Because the compiler, not the programmer, is arranging for the mutual exclusion, it is much less likely that something will go wrong. In any event, the person writing the monitor does not have to be aware of how the compiler arranges for mutual exclusion. It is sufficient to know that by turning all the critical regions into monitor procedures, no two processes will ever execute their critical regions at the same time.

Although monitors provide an easy way to achieve mutual exclusion, as we have seen above, that is not enough. We also need a way for processes to block when they cannot proceed. In the producer-consumer problem, it is easy enough to put all the tests for buffer-full and buffer-empty in monitor procedures, but how should the producer block when it finds the buffer full?

The solution lies in the introduction of **condition variables**, along with two operations on them, wait and signal. When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a wait on some condition variable, say, *full*. This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now.

This other process, for example, the consumer, can wake up its sleeping partner by doing a signal on the condition variable that its partner is waiting on. To avoid having two active processes in the monitor at the same time, we need a rule telling what happens after a signal. Hoare proposed letting the newly awakened process run, suspending the other one. Brinch Hansen proposed finessing the problem by requiring that a process doing a signal *must* exit the monitor immediately. In other words, a signal statement may appear only as the final statement in a monitor procedure. We will use Brinch Hansen's proposal because it is conceptually simpler and is also easier to implement. If a signal is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler, is revived.

As an aside, there is also a third solution, not proposed by either Hoare or Brinch Hansen. This is to let the signaler continue to run and allow the waiting process to start running only after the signaler has exited the monitor.

Condition variables are not counters. They do not accumulate signals for later use the way semaphores do. Thus if a condition variable is signaled with no one waiting on it, the signal is lost forever. In other words, the wait must come before the signal. This rule makes the implementation much simpler. In practice it is not a problem because it is easy to keep track of the state of each process with variables, if need be. A process that might otherwise do a signal can see that this operation is not necessary by looking at the variables.

A skeleton of the producer-consumer problem with monitors is given in Fig. 2-27 in an imaginary language, Pidgin Pascal. The advantage of using Pidgin Pascal here is that it is pure and simple and follows the Hoare/Brinch Hansen model exactly.

```
monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;

    count := 0;
end monitor;

procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

Figure 2-27. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has *N* slots.

You may be thinking that the operations wait and signal look similar to sleep and wakeup, which we saw earlier had fatal race conditions. They *are* very similar, but with one crucial difference: sleep and wakeup failed because while one process was trying to go to sleep, the other one was trying to wake it up. With monitors, that cannot happen. The automatic mutual exclusion on monitor procedures guarantees that if, say, the producer inside a monitor procedure discovers that the buffer is full, it will be able to complete the wait operation without having to worry about the possibility that the scheduler may switch to the consumer just before the wait completes. The consumer will not even be let into the monitor at all until the wait is finished and the producer has been marked as no longer runnable.

Although Pidgin Pascal is an imaginary language, some real programming languages also support monitors, although not always in the form designed by Hoare and Brinch Hansen. One such language is Java. Java is an object-oriented language that supports user-level threads and also allows methods (procedures) to be grouped together into classes. By adding the keyword synchronized to a method declaration, Java guarantees that once any thread has started executing that method, no other thread will be allowed to start executing any other synchronized method in that class.

A solution to the producer-consumer problem using monitors in Java is given in Fig. 2-28. The solution consists of four classes. The outer class, *ProducerConsumer*, creates and starts two threads, *p* and *c*. The second and third classes, *producer* and *consumer*, respectively, contain the code for the producer and consumer. Finally, the class *our_monitor*, is the monitor. It contains two synchronized threads that are used for actually inserting items into the shared buffer and taking them out. Unlike in the previous examples, we have finally shown the full code of *insert* and *remove* here.

The producer and consumer threads are functionally identical to their counterparts in all our previous examples. The producer has an infinite loop generating data and putting it into the common buffer. The consumer has an equally infinite loop taking data out of the common buffer and doing some fun thing with it.

The interesting part of this program is the class *our_monitor*, which contains the buffer, the administration variables, and two synchronized methods. When the producer is active inside *insert*, it knows for sure that the consumer cannot be active inside *remove*, making it safe to update the variables and the buffer without fear of race conditions. The variable *count* keeps track of how many items are in the buffer. It can take on any value from 0 through and including $N - 1$. The variable *lo* is the index of the buffer slot where the next item is to be fetched. Similarly, *hi* is the index of the buffer slot where the next item is to be placed. It is permitted that $lo = hi$, which means either that 0 items or N items are in the buffer. The value of *count* tells which case holds.

Synchronized methods in Java differ from classical monitors in an essential way: Java does not have condition variables. Instead, it offers two procedures,

wait and *notify* that are the equivalent of *sleep* and *wakeup* except that when they are used inside synchronized methods, they are not subject to race conditions. In theory, the method *wait* can be interrupted, which is what the code surrounding it is all about. Java requires that the exception handling be made explicit. For our purposes, just imagine that *go_to_sleep* is the way to go to sleep.

By making the mutual exclusion of critical regions automatic, monitors make parallel programming much less error-prone than with semaphores. Still, they too have some drawbacks. It is not for nothing that our two examples of monitors were in Pidgin Pascal and Java instead of C, as are the other examples in this book. As we said earlier, monitors are a programming language concept. The compiler must recognize them and arrange for the mutual exclusion somehow. C, Pascal, and most other languages do not have monitors, so it is unreasonable to expect their compilers to enforce any mutual exclusion rules. In fact, how could the compiler even know which procedures were in monitors and which were not?

These same languages do not have semaphores either, but adding semaphores is easy: All you need to do is add two short assembly code routines to the library to issue the up and down system calls. The compilers do not even have to know that they exist. Of course, the operating systems have to know about the semaphores, but at least if you have a semaphore-based operating system, you can still write the user programs for it in C or C++ (or even assembly language if you are masochistic enough). With monitors, you need a language that has them built in.

Another problem with monitors, and also with semaphores, is that they were designed for solving the mutual exclusion problem on one or more CPUs that all have access to a common memory. By putting the semaphores in the shared memory and protecting them with TSL instructions, we can avoid races. When we go to a distributed system consisting of multiple CPUs, each with its own private memory, connected by a local area network, these primitives become inapplicable. The conclusion is that semaphores are too low level and monitors are not usable except in a few programming languages. Also, none of the primitives provide for information exchange between machines. Something else is needed.

2.3.8 Message Passing

That something else is **message passing**. This method of interprocess communication uses two primitives, send and receive, which, like semaphores and unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as

```
send(destination, &message);
```

and

```
receive(source, &message);
```

The former call sends a message to a given destination and the latter one receives

```
public class ProducerConsumer {
    static final int N = 100;      // constant giving the buffer size
    static producer p = new producer( );   // instantiate a new producer thread
    static consumer c = new consumer( ); // instantiate a new consumer thread
    static our_monitor mon = new our_monitor( );      // instantiate a new monitor

    public static void main(String args[ ]) {
      p.start( );      // start the producer thread
      c.start( );      // start the consumer thread
    }

    static class producer extends Thread {
      public void run( ) {// run method contains the thread code
        int item;
        while (true) {     // producer loop
          item = produce_item( );
          mon.insert(item);
        }
      }
      private int produce_item( ) { ... }      // actually produce
    }

    static class consumer extends Thread {
      public void run( ) {run method contains the thread code
        int item;
        while (true) {     // consumer loop
          item = mon.remove( );
          consume_item (item);
        }
      }
      private void consume_item(int item) { ... }// actually consume
    }

    static class our_monitor {  // this is a monitor
      private int buffer[ ] = new int[N];
      private int count = 0, lo = 0, hi = 0;  // counters and indices

      public synchronized void insert(int val) {
        if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
        buffer [hi] = val; // insert an item into the buffer
        hi = (hi + 1) % N;       // slot to place next item in
        count = count + 1;       // one more item in the buffer now
        if (count == 1) notify( );       // if consumer was sleeping, wake it up
      }

      public synchronized int remove( ) {
        int val;
        if (count == 0) go_to_sleep( );     // if the buffer is empty, go to sleep
        val = buffer [lo]; // fetch an item from the buffer
        lo = (lo + 1) % N;       // slot to fetch next item from
        count = count – 1;       // one few items in the buffer
        if (count == N – 1) notify( ); // if producer was sleeping, wake it up
        return val;
      }
      private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
    }
}
```

Figure 2-28. A solution to the producer-consumer problem in Java.

a message from a given source (or from *ANY*, if the receiver does not care). If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code.

Design Issues for Message Passing Systems

Message passing systems have many challenging problems and design issues that do not arise with semaphores or monitors, especially if the communicating processes are on different machines connected by a network. For example, messages can be lost by the network. To guard against lost messages, the sender and receiver can agree that as soon as a message has been received, the receiver will send back a special **acknowledgement** message. If the sender has not received the acknowledgement within a certain time interval, it retransmits the message.

Now consider what happens if the message itself is received correctly, but the acknowledgement is lost. The sender will retransmit the message, so the receiver will get it twice. It is essential that the receiver be able to distinguish a new message from the retransmission of an old one. Usually, this problem is solved by putting consecutive sequence numbers in each original message. If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a duplicate that can be ignored. Successfully communicating in the face of unreliable message passing is a major part of the study of computer networks. For more information, see (Tanenbaum, 1996).

Message systems also have to deal with the question of how processes are named, so that the process specified in a send or receive call is unambiguous. **Authentication** is also an issue in message systems: how can the client tell that he is communicating with the real file server, and not with an imposter?

At the other end of the spectrum, there are also design issues that are important when the sender and receiver are on the same machine. One of these is performance. Copying messages from one process to another is always slower than doing a semaphore operation or entering a monitor. Much work has gone into making message passing efficient. Cheriton (1984), for example, suggested limiting message size to what will fit in the machine's registers, and then doing message passing using the registers.

The Producer-Consumer Problem with Message Passing

Now let us see how the producer-consumer problem can be solved with message passing and no shared memory. A solution is given in Fig. 2-29. We assume that all messages are the same size and that messages sent but not yet received are buffered automatically by the operating system. In this solution, a total of N messages is used, analogous to the N slots in a shared memory buffer. The consumer

starts out by sending *N* empty messages to the producer. Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one. In this way, the total number of messages in the system remains constant in time, so they can be stored in a given amount of memory known in advance.

If the producer works faster than the consumer, all the messages will end up full, waiting for the consumer; the producer will be blocked, waiting for an empty to come back. If the consumer works faster, then the reverse happens: all the messages will be empties waiting for the producer to fill them up; the consumer will be blocked, waiting for a full message.

```
#define N 100                          /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                         /* message buffer */

    while (TRUE) {
        item = produce_item( );        /* generate something to put in buffer */
        receive(consumer, &m);         /* wait for an empty to arrive */
        build_message(&m, item);       /* construct a message to send */
        send(consumer, &m);            /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
    while (TRUE) {
        receive(producer, &m);         /* get message containing item */
        item = extract_item(&m);       /* extract item from message */
        send(producer, &m);            /* send back empty reply */
        consume_item(item);            /* do something with the item */
    }
}
```

Figure 2-29. The producer-consumer problem with *N* messages.

Many variants are possible with message passing. For starters, let us look at how messages are addressed. One way is to assign each process a unique address and have messages be addressed to processes. A different way is to invent a new data structure, called a **mailbox**. A mailbox is a place to buffer a certain number of messages, typically specified when the mailbox is created. When mailboxes

are used, the address parameters in the send and receive calls are mailboxes, not processes. When a process tries to send to a mailbox that is full, it is suspended until a message is removed from that mailbox, making room for a new one.

For the producer-consumer problem, both the producer and consumer would create mailboxes large enough to hold *N* messages. The producer would send messages containing data to the consumer's mailbox, and the consumer would send empty messages to the producer's mailbox. When mailboxes are used, the buffering mechanism is clear: the destination mailbox holds messages that have been sent to the destination process but have not yet been accepted.

The other extreme from having mailboxes is to eliminate all buffering. When this approach is followed, if the send is done before the receive, the sending process is blocked until the receive happens, at which time the message can be copied directly from the sender to the receiver, with no intermediate buffering. Similarly, if the receive is done first, the receiver is blocked until a send happens. This strategy is often known as a **rendezvous**. It is easier to implement than a buffered message scheme but is less flexible since the sender and receiver are forced to run in lockstep.

Message passing is commonly used in parallel programming systems. One well-known message-passing system, for example, is **MPI (Message-Passing Interface)**. It is widely used for scientific computing. For more information about it, see for example (Gropp et al., 1994; and Snir et al., 1996).

2.3.9 Barriers

Our last synchronization mechanism is intended for groups of processes rather than two-process producer-consumer type situations. Some applications are divided into phases and have the rule that no process may proceed into the next phase until all processes are ready to proceed to the next phase. This behavior may be achieved by placing a **barrier** at the end of each phase. When a process reaches the barrier, it is blocked until all processes have reached the barrier. The operation of a barrier is illustrated in Fig. 2-30.

In Fig. 2-30(a) we see four processes approaching a barrier. What this means is that they are just computing and have not reached the end of the current phase yet. After a while, the first process finishes all the computing required of it during the first phase. It then executes the barrier primitive, generally by calling a library procedure. The process is then suspended. A little later, a second and then a third process finish the first phase and also execute the barrier primitive. This situation is illustrated in Fig. 2-30(b). Finally, when the last process, *C*, hits the barrier, all the processes are released, as shown in Fig. 2-30(c).

As an example of a problem requiring barriers, consider a typical relaxation problem in physics or engineering. There is typically a matrix that contains some initial values. The values might represent temperatures at various points on a

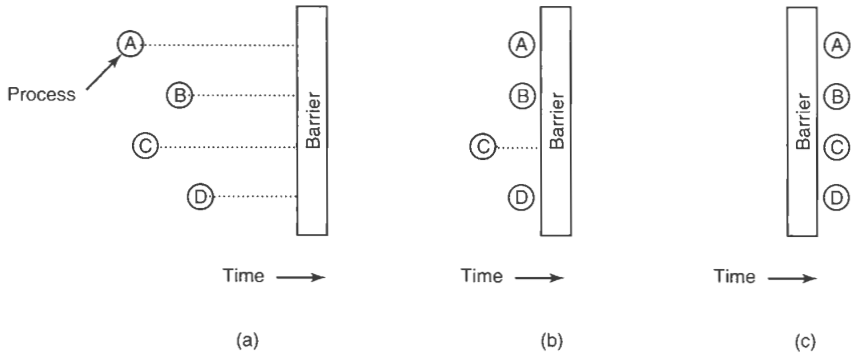


Figure 2-30. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

sheet of metal. The idea might be to calculate how long it takes for the effect of a flame placed at one corner to propagate throughout the sheet.

Starting with the current values, a transformation is applied to the matrix to get the second version of the matrix, for example, by applying the laws of thermodynamics to see what all the temperatures are ΔT later. Then the processes is repeated over and over, giving the temperatures at the sample points as a function of time as the sheet heats up. The algorithm thus produces a series of matrices over time.

Now imagine that the matrix is very large (say, 1 million by 1 million), so that parallel processes are needed (possibly on a multiprocessor) to speed up the calculation. Different processes work on different parts of the matrix, calculating the new matrix elements from the old ones according to the laws of physics. However, no process may start on iteration $n + 1$ until iteration n is complete, that is, until all processes have finished their current work. The way to achieve this goal is to program each process to execute a barrier operation after it has finished its part of the current iteration. When all of them are done, the new matrix (the input to the next iteration) will be finished, and all processes will be simultaneously released to start the next iteration.

2.4 CLASSICAL IPC PROBLEMS

The operating systems literature is full of interesting problems that have been widely discussed and analyzed using a variety of synchronization methods. In the following sections we will examine three of the better-known problems.

2.4.1 The Dining Philosophers Problem

In 1965, Dijkstra posed and solved a synchronization problem he called the **dining philosophers problem**. Since that time, everyone inventing yet another synchronization primitive has felt obligated to demonstrate how wonderful the new primitive is by showing how elegantly it solves the dining philosophers problem. The problem can be stated quite simply as follows. Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The layout of the table is illustrated in Fig. 2-31.

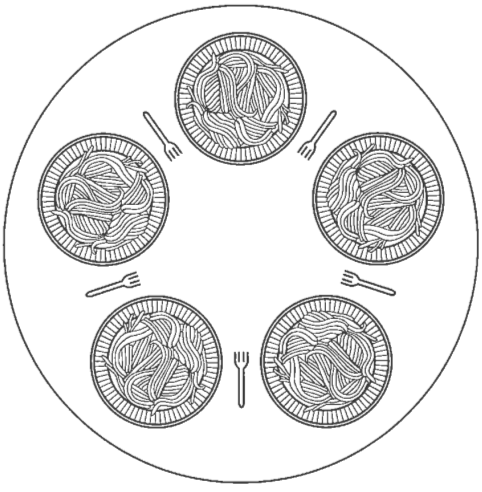


Figure 2-31. Lunch time in the Philosophy Department.

The life of a philosopher consists of alternate periods of eating and thinking. (This is something of an abstraction, even for philosophers, but the other activities are irrelevant here.) When a philosopher gets hungry, she tries to acquire her left and right fork, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think. The key question is: Can you write a program for each philosopher that does what it is supposed to do and never gets stuck? (It has been pointed out that the two-fork requirement is somewhat artificial; perhaps we should switch from Italian food to Chinese food, substituting rice for spaghetti and chopsticks for forks.)

Figure 2-32 shows the obvious solution. The procedure *take_fork* waits until the specified fork is available and then seizes it. Unfortunately, the obvious solution is wrong. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

We could modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down

```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                    /* philosopher is thinking */
        take_fork(i);                /* take left fork */
        take_fork((i+1) % N);        /* take right fork; % is modulo operator */
        eat( );                      /* yum-yum, spaghetti */
        put_fork(i);                 /* put left fork back on the table */
        put_fork((i+1) % N);         /* put right fork back on the table */
    }
}
```

Figure 2-32. A nonsolution to the dining philosophers problem.

the left one, waits for some time, and then repeats the whole process. This proposal too, fails, although for a different reason. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, picking up their left forks again simultaneously, and so on, forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called **starvation**. (It is called starvation even when the problem does not occur in an Italian or a Chinese restaurant.)

Now you might think, "If the philosophers would just wait a random time instead of the same time after failing to acquire the right-hand fork, the chance that everything would continue in lockstep for even an hour is very small." This observation is true, and in nearly all applications trying again later is not a problem. For example, in the popular Ethernet local area network, if two computers send a packet at the same time, each one waits a random time and tries again; in practice this solution works fine. However, in a few applications one would prefer a solution that always works and cannot fail due to an unlikely series of random numbers. Think about safety control in a nuclear power plant.

One improvement to Fig. 2-32 that has no deadlock and no starvation is to protect the five statements following the call to *think* by a binary semaphore. Before starting to acquire forks, a philosopher would do a down on *mutex*. After replacing the forks, she would do an up on *mutex*. From a theoretical viewpoint, this solution is adequate. From a practical one, it has a performance bug: only one philosopher can be eating at any instant. With five forks available, we should be able to allow two philosophers to eat at the same time.

The solution presented in Fig. 2-33 is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, *state*, to keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may move only into eating state if neither neighbor

```
#define N            5          /* number of philosophers */
#define LEFT         (i+N-1)%N  /* number of i's left neighbor */
#define RIGHT        (i+1)%N    /* number of i's right neighbor */
#define THINKING     0          /* philosopher is thinking */
#define HUNGRY       1          /* philosopher is trying to get forks */
#define EATING       2          /* philosopher is eating */
typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];                 /* one semaphore per philosopher */

void philosopher(int i)         /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {              /* repeat forever */
        think( );               /* philosopher is thinking */
        take_forks(i);          /* acquire two forks or block */
        eat( );                 /* yum-yum, spaghetti */
        put_forks(i);           /* put both forks back on table */
    }
}

void take_forks(int i)          /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = HUNGRY;          /* record fact that philosopher i is hungry */
    test(i);                    /* try to acquire 2 forks */
    up(&mutex);                 /* exit critical region */
    down(&s[i]);                /* block if forks were not acquired */
}

void put_forks(i)               /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = THINKING;        /* philosopher has finished eating */
    test(LEFT);                 /* see if left neighbor can now eat */
    test(RIGHT);                /* see if right neighbor can now eat */
    up(&mutex);                 /* exit critical region */
}

void test(i)                    /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Figure 2-33. A solution to the dining philosophers problem.

is eating. Philosopher *i*'s neighbors are defined by the macros *LEFT* and *RIGHT*. In other words, if *i* is 2, *LEFT* is 1 and *RIGHT* is 3.

The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy. Note that each process runs the procedure *philosopher* as its main code, but the other procedures, *take_forks*, *put_forks*, and *test* are ordinary procedures and not separate processes.

2.4.2 The Readers and Writers Problem

The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem (Courtois et al., 1971), which models access to a database. Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers. The question is how do you program the readers and the writers? One solution is shown in Fig. 2-34.

In this solution, the first reader to get access to the database does a down on the semaphore *db*. Subsequent readers merely increment a counter, *rc*. As readers leave, they decrement the counter and the last one out does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

The solution presented here implicitly contains a subtle decision that is worth commenting on. Suppose that while a reader is using the database, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. A third and subsequent readers can also be admitted if they come along.

Now suppose that a writer comes along. The writer cannot be admitted to the database, since writers must have exclusive access, so the writer is suspended. Later, additional readers show up. As long as at least one reader is still active, subsequent readers are admitted. As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive. The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 seconds, and each reader takes 5 seconds to do its work, the writer will never get in.

To prevent this situation, the program could be written slightly differently: when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately. In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less concurrency and thus lower performance. Courtois et al. present a solution that gives priority to writers. For details, we refer you to the paper.

```
typedef int semaphore;              /* use your imagination */
semaphore mutex = 1;                /* controls access to 'rc' */
semaphore db = 1;                   /* controls access to the database */
int rc = 0;                         /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                  /* repeat forever */
        down(&mutex);               /* get exclusive access to 'rc' */
        rc = rc + 1;                /* one reader more now */
        if (rc == 1) down(&db);     /* if this is the first reader ... */
        up(&mutex);                 /* release exclusive access to 'rc' */
        read_data_base( );          /* access the data */
        down(&mutex);               /* get exclusive access to 'rc' */
        rc = rc – 1;                /* one reader fewer now */
        if (rc == 0) up(&db);       /* if this is the last reader ... */
        up(&mutex);                 /* release exclusive access to 'rc' */
        use_data_read( );           /* noncritical region */
    }
}


void writer(void)
{
    while (TRUE) {                  /* repeat forever */
        think_up_data( );           /* noncritical region */
        down(&db);                  /* get exclusive access */
        write_data_base( );         /* update the data */
        up(&db);                    /* release exclusive access */
    }
}
```

Figure 2-34. A solution to the readers and writers problem.

2.4.3 The Sleeping Barber Problem

Another classical IPC problem takes place in a barber shop. The barber shop has one barber, one barber chair, and *n* chairs for waiting customers, if any, to sit on. If there are no customers present, the barber sits down in the barber chair and falls asleep, as illustrated in Fig. 2-35. When a customer arrives, he has to wake up the sleeping barber. If additional customers arrive while the barber is cutting a customer's hair, they either sit down (if there are empty chairs) or leave the shop (if all chairs are full). The problem is to program the barber and the customers

without getting into race conditions. This problem is similar to various queueing situations, such as a multiperson helpdesk with a computerized call waiting system for holding a limited number of incoming calls.

Figure 2-35. The sleeping barber.

Our solution uses three semaphores: *customers*, which counts waiting customers (excluding the customer in the barber chair, who is not waiting), *barbers*, the number of barbers (0 or 1) who are idle, waiting for customers, and *mutex*, which is used for mutual exclusion. We also need a variable, *waiting*, which also counts the waiting customers. It is essentially a copy of *customers*. The reason for having *waiting* is that there is no way to read the current value of a semaphore. In this solution, a customer entering the shop has to count the number of waiting customers. If it is less than the number of chairs, he stays; otherwise, he leaves.

Our solution is shown in Fig. 2-36. When the barber shows up for work in the morning, he executes the procedure *barber*, causing him to block on the semaphore *customers* because it is initially 0. The barber then goes to sleep, as shown in Fig. 2-35. He stays asleep until the first customer shows up.

```
#define CHAIRS 5                    /* # chairs for waiting customers */

typedef int semaphore;              /* use your imagination */

semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;              /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* go to sleep if # of customers is 0 */
        down(&mutex);               /* acquire access to 'waiting' */
        waiting = waiting – 1;      /* decrement count of waiting customers */
        up(&barbers);               /* one barber is now ready to cut hair */
        up(&mutex);                 /* release 'waiting' */
        cut_hair( );                /* cut hair (outside critical region) */
    }
}


void customer(void)
{
    down(&mutex);                   /* enter critical region */
    if (waiting < CHAIRS) {         /* if there are no free chairs, leave */
        waiting = waiting + 1;      /* increment count of waiting customers */
        up(&customers);             /* wake up barber if necessary */
        up(&mutex);                 /* release access to 'waiting' */
        down(&barbers);             /* go to sleep if # of free barbers is 0 */
        get_haircut( );             /* be seated and be serviced */
    } else {
        up(&mutex);                 /* shop is full; do not wait */
    }
}
```

Figure 2-36. A solution to the sleeping barber problem.

When a customer arrives, he executes *customer*, starting by acquiring *mutex* to enter a critical region. If another customer enters shortly thereafter, the second one will not be able to do anything until the first one has released *mutex*. The customer then checks to see if the number of waiting customers is less than the number of chairs. If not, he releases *mutex* and leaves without a haircut.

If there is an available chair, the customer increments the integer variable, *waiting*. Then he does an up on the semaphore *customers*, thus waking up the

barber. At this point, the customer and barber are both awake. When the customer releases *mutex*, the barber grabs it, does some housekeeping, and begins the haircut.

When the haircut is over, the customer exits the procedure and leaves the shop. Unlike our earlier examples, there is no loop for the customer because each one gets only one haircut. The barber loops, however, to try to get the next customer. If one is present, another haircut is given. If not, the barber goes to sleep.

As an aside, it is worth pointing out that although the readers and writers and sleeping barber problems do not involve data transfer, they are still belong to the area of IPC because they involve synchronization between multiple processes.

2.5 SCHEDULING

When a computer is multiprogrammed, it frequently has multiple processes competing for the CPU at the same time. This situation occurs whenever two or more processes are simultaneously in the ready state. If only one CPU is available, a choice has to be made which process to run next. The part of the operating system that makes the choice is called the **scheduler** and the algorithm it uses is called the **scheduling algorithm**. These topics form the subject matter of the following sections.

Many of the same issues that apply to process scheduling also apply to thread scheduling, although some are different. Initially we will focus on process scheduling. Later on we will explicitly look at thread scheduling.

2.5.1 Introduction to Scheduling

Back in the old days of batch systems with input in the form of card images on a magnetic tape, the scheduling algorithm was simple: just run the next job on the tape. With timesharing systems, the scheduling algorithm became more complex because there were generally multiple users waiting for service. Some mainframes still combine batch and timesharing service, requiring the scheduler to decide whether a batch job or an interactive user at a terminal should go next. (As an aside, a batch job may be a request to run multiple programs in succession, but for this section, we will just assume it is a request to run a single program.) Because CPU time is a scarce resource on these machines, a good scheduler can make a big difference in perceived performance and user satisfaction. Consequently, a great deal of work has gone into devising clever and efficient scheduling algorithms.

With the advent of personal computers, the situation changed in two ways. First, most of the time there is only one active process. A user entering a document on a word processor is unlikely to be simultaneously compiling a program in the background. When the user types a command to the word processor, the

scheduler does not have to do much work to figure out which process to run—the word processor is the only candidate.

Second, computers have gotten so much faster over the years that the CPU is rarely a scarce resource any more. Most programs for personal computers are limited by the rate at which the user can present input (by typing or clicking), not by the rate the CPU can process it. Even compilations, a major sink of CPU cycles in the past, take just a few seconds at most nowadays. Even when two programs are actually running at once, such as a word processor and a spreadsheet, it hardly matters which goes first since the user is probably waiting for both of them to finish. As a consequence, scheduling does not matter much on simple PCs. [Of course, there are applications that practically eat the CPU alive: rendering one hour of high-resolution video may require industrial-strength image processing on each of 108,000 frames in NTSC (90,000 in PAL), but these applications are the exception rather than the rule.]

When we turn to high-end networked workstations and servers, the situation changes. Here multiple processes often do compete for the CPU, so scheduling matters again. For example, when the CPU has to choose between running a process that updates the screen after a user has closed a window and running a process that sends out queued email, it makes a huge difference in the perceived response. If closing the window were to take 2 sec while the email was being sent, the user would probably regard the system as extremely sluggish, whereas having the email delayed by 2 sec would not even be noticed. In this case, process scheduling matters very much.

In addition to picking the right process to run, the scheduler also has to worry about making efficient use of the CPU because process switching is expensive. To start with, a switch from user mode to kernel mode must occur. Then the state of the current process must be saved, including storing its registers in the process table so they can be reloaded later. In many systems, the memory map (e.g., memory reference bits in the page table) must be saved as well. Next a new process must be selected by running the scheduling algorithm. After that, the MMU must be reloaded with the memory map of the new process. Finally, the new process must be started. In addition to all that, the process switch usually invalidates the entire memory cache, forcing it to be dynamically reloaded from the main memory twice (upon entering the kernel and upon leaving it). All in all, doing too many process switches per second can chew up a substantial amount of CPU time, so caution is advised.

Process Behavior

Nearly all processes alternate bursts of computing with (disk) I/O requests, as shown in Fig. 2-37. Typically the CPU runs for a while without stopping, then a system call is made to read from a file or write to a file. When the system call completes, the CPU computes again until it needs more data or has to write more

data, and so on. Note that some I/O activities count as computing. For example, when the CPU copies bits to a video RAM to update the screen, it is computing, not doing I/O, because the CPU is in use. I/O in this sense is when a process enters the blocked state waiting for an external device to complete its work.

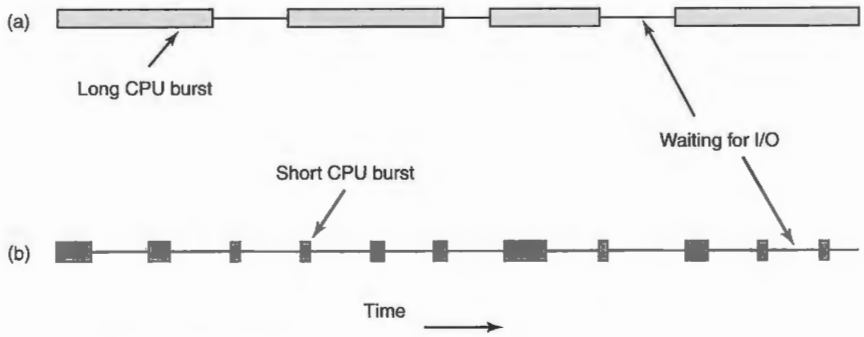


Figure 2-37. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

The important thing to notice about Fig. 2-37 is that some processes, such as the one in Fig. 2-37(a), spend most of their time computing, while others, such as the one in Fig. 2-37(b), spend most of their time waiting for I/O. The former are called **compute-bound**; the latter are called **I/O-bound**. Compute-bound processes typically have long CPU bursts and thus infrequent I/O waits, whereas I/O-bound processes have short CPU bursts and thus frequent I/O waits. Note that the key factor is the length of the CPU burst, not the length of the I/O burst. I/O-bound processes are I/O bound because they do not compute much between I/O requests, not because they have especially long I/O requests. It takes the same time to read a disk block no matter how much or how little time it takes to process the data after they arrive.

It is worth noting that as CPUs get faster, processes tend to get more I/O-bound. This effect occurs because CPUs are improving much faster than disks. As a consequence, the scheduling of I/O-bound processes is likely to become a more important subject in the future. The basic idea here is that if an I/O-bound process wants to run, it should get a chance quickly so it can issue its disk request and keep the disk busy.

When to Schedule

A key issue related to scheduling is when to make scheduling decisions. It turns out that there are a variety of situations in which scheduling is needed. First, when a new process is created, a decision needs to be made whether to run the

parent process or the child process. Since both processes are in ready state, it is a normal scheduling decision and it can go either way, that is, the scheduler can legitimately choose to run either the parent or the child next.

Second, a scheduling decision must be made when a process exits. That process can no longer run (since it no longer exists), so some other process must be chosen from the set of ready processes. If no process is ready, a system-supplied idle process is normally run.

Third, when a process blocks on I/O, on a semaphore, or for some other reason, another process has to be selected to run. Sometimes the reason for blocking may play a role in the choice. For example, if *A* is an important process and it is waiting for *B* to exit its critical region, letting *B* run next will allow it to exit its critical region and thus let *A* continue. The trouble, however, is that the scheduler generally does not have the necessary information to take this dependency into account.

Fourth, when an I/O interrupt occurs, a scheduling decision may be made. If the interrupt came from an I/O device that has now completed its work, some process that was blocked waiting for the I/O may now be ready to run. It is up to the scheduler to decide if the newly ready process should be run, if the process that was running at the time of the interrupt should continue running, or if some third process should run.

If a hardware clock provides periodic interrupts at 50 Hz, 60 Hz, or some other frequency, a scheduling decision can be made at each clock interrupt or at every *k*-th clock interrupt. Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts. A **nonpreemptive** scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or until it voluntarily releases the CPU. Even if it runs for hours, it will not be forceably suspended. In effect, no scheduling decisions are made during clock interrupts. After clock interrupt processing has been completed, the process that was running before the interrupt is always resumed.

In contrast, a **preemptive** scheduling algorithm picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available). Doing preemptive scheduling requires having a clock interrupt occur at the end of the time interval to give control of the CPU back to the scheduler. If no clock is available, nonpreemptive scheduling is the only option.

Categories of Scheduling Algorithms

Not surprisingly, in different environments different scheduling algorithms are needed. This situation arises because different application areas (and different kinds of operating systems) have different goals. In other words, what the sched-

uler should optimize for is not the same in all systems. Three environments worth distinguishing are

1. Batch.
2. Interactive.
3. Real time.

In batch systems, there are no users impatiently waiting at their terminals for a quick response. Consequently, nonpreemptive algorithms, or preemptive algorithms with long time periods for each process are often acceptable. This approach reduces process switches and thus improves performance.

In an environment with interactive users, preemption is essential to keep one process from hogging the CPU and denying service to the others. Even if no process intentionally ran forever, due to a program bug, one process might shut out all the others indefinitely. Preemption is needed to prevent this behavior.

In systems with real-time constraints, preemption is, oddly enough, sometimes not needed because the processes know that they may not run for long periods of time and usually do their work and block quickly. The difference with interactive systems is that real-time systems run only programs that are intended to further the application at hand. Interactive systems are general purpose and may run arbitrary programs that are not cooperative or even malicious.

Scheduling Algorithm Goals

In order to design a scheduling algorithm, it is necessary to have some idea of what a good algorithm should do. Some goals depend on the environment (batch, interactive, or real time), but there are also some that are desirable in all cases. Some goals are listed in Fig. 2-38. We will discuss these in turn below.

Under all circumstances, fairness is important. Comparable processes should get comparable service. Giving one process much more CPU time than an equivalent one is not fair. Of course, different categories of processes may be treated very differently. Think of safety control and doing the payroll at a nuclear reactor's computer center.

Somewhat related to fairness is enforcing the system's policies. If the local policy is that safety control processes get to run whenever they want to, even if it means the payroll is 30 sec late, the scheduler has to make sure this policy is enforced.

Another general goal is keeping all parts of the system busy when possible. If the CPU and all the I/O devices can be kept running all the time, more work gets done per second than if some of the components are idle. In a batch system, for example, the scheduler has control of which jobs are brought into memory to run. Having some CPU-bound processes and some I/O-bound processes in memory together is a better idea than first loading and running all the CPU-bound jobs and

All systems
- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

Batch systems
- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

Interactive systems
- Response time - respond to requests quickly
- Proportionality - meet users' expectations

Real-time systems
- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

Figure 2-38. Some goals of the scheduling algorithm under different circumstances.

then when they are finished loading and running all the I/O-bound jobs. If the latter strategy is used, when the CPU-bound processes are running, they will fight for the CPU and the disk will be idle. Later, when the I/O-bound jobs come in, they will fight for the disk and the CPU will be idle. Better to keep the whole system running at once by a careful mix of processes.

The managers of large computer centers that run many batch jobs typically look at three metrics to see how well their systems are performing: throughput, turnaround time, and CPU utilization. **Throughput** is the number of jobs per hour that the system completes. All things considered, finishing 50 jobs per hour is better than finishing 40 jobs per hour. **Turnaround time** is the statistically average time from the moment that a batch job is submitted until the moment it is completed. It measures how long the average user has to wait for the output. Here the rule is: Small is Beautiful.

A scheduling algorithm that maximizes throughput may not necessarily minimize turnaround time. For example, given a mix of short jobs and long jobs, a scheduler that always ran short jobs and never ran long jobs might achieve an excellent throughput (many short jobs per hour) but at the expense of a terrible turnaround time for the long jobs. If short jobs kept arriving at a steady rate, the long jobs might never run, making the mean turnaround time infinite while achieving a high throughput.

CPU utilization is also an issue with batch systems because on the big mainframes where batch systems run, the CPU is still a major expense. Thus computer center managers feel guilty when it is not running all the time. Actually though, this is not such a good metric. What really matters is how many jobs per hour

come out of the system (throughput) and how long it takes to get a job back (turnaround time). Using CPU utilization as a metric is like rating cars based on how many times per hour the engine turns over.

For interactive systems, especially timesharing systems and servers, different goals apply. The most important one is to minimize **response time**, that is the time between issuing a command and getting the result. On a personal computer where a background process is running (for example, reading and storing email from the network), a user request to start a program or open a file should take precedence over the background work. Having all interactive requests go first will be perceived as good service.

A somewhat related issue is what might be called **proportionality**. Users have an inherent (but often incorrect) idea of how long things should take. When a request that is perceived as complex takes a long time, users accept that, but when a request that is perceived as simple takes a long time, users get irritated. For example, if clicking on a icon that calls up an Internet provider using an analog modem takes 45 seconds to establish a connection, the user will probably accept that as a fact of life. On the other hand, if clicking on an icon that breaks the connection takes 45 seconds, the user will probably be swearing a blue streak by the 30-sec mark and frothing at the mouth by 45 sec. This behavior is due to the common user perception that placing a phone call and getting a connection is *supposed* to take a lot longer than just hanging up. In some cases (such as this one), the scheduler cannot do anything about the response time, but in other cases it can, especially when the delay is due to a poor choice of process order.

Real-time systems have different properties than interactive systems, and thus different scheduling goals. They are characterized by having deadlines that must or at least should be met. For example, if a computer is controlling a device that produces data at a regular rate, failure to run the data-collection process on time may result in lost data. Thus the foremost need in a real-time system is meeting all (or most) deadlines.

In some real-time systems, especially those involving multimedia, predictability is important. Missing an occasional deadline is not fatal, but if the audio process runs too erratically, the sound quality will deteriorate rapidly. Video is also an issue, but the ear is much more sensitive to jitter than the eye. To avoid this problem, process scheduling must be highly predictable and regular. We will study batch and interactive scheduling algorithms in this chapter but defer most of our study of real-time scheduling until we come to multimedia operating systems in Chap. 7.

2.5.2 Scheduling in Batch Systems

It is now time to turn from general scheduling issues to specific scheduling algorithms. In this section we will look at algorithms used in batch systems. In the following ones we will examine interactive and real-time systems. It is worth

pointing out that some algorithms are used in both batch and interactive systems. We will study these later. Here we will focus on algorithms that are only suitable in batch systems.

First-Come First-Served

Probably the simplest of all scheduling algorithms is nonpreemptive **first-come first-served**. With this algorithm, processes are assigned the CPU in the order they request it. Basically, there is a single queue of ready processes. When the first job enters the system from the outside in the morning, it is started immediately and allowed to run as long as it wants to. As other jobs come in, they are put onto the end of the queue. When the running process blocks, the first process on the queue is run next. When a blocked process becomes ready, like a newly arrived job, it is put on the end of the queue.

The great strength of this algorithm is that it is easy to understand and equally easy to program. It is also fair in the same sense that allocating scarce sports or concert tickets to people who are willing to stand on line starting at 2 A.M. is fair. With this algorithm, a single linked list keeps track of all ready processes. Picking a process to run just requires removing one from the front of the queue. Adding a new job or unblocked process just requires attaching it to the end of the queue. What could be simpler?

Unfortunately, first-come first-served also has a powerful disadvantage. Suppose that there is one compute-bound process that runs for 1 sec at a time and many I/O-bound processes that use little CPU time but each have to perform 1000 disk reads to complete. The compute-bound process runs for 1 sec, then it reads a disk block. All the I/O processes now run and start disk reads. When the compute-bound process gets its disk block, it runs for another 1 sec, followed by all the I/O-bound processes in quick succession.

The net result is that each I/O-bound process gets to read 1 block per second and will take 1000 sec to finish. With a scheduling algorithm that preempted the compute-bound process every 10 msec, the I/O-bound processes would finish in 10 sec instead of 1000 sec, and without slowing down the compute-bound process very much.

Shortest Job First

Now let us look at another nonpreemptive batch algorithm that assumes the run times are known in advance. In an insurance company, for example, people can predict quite accurately how long it will take to run a batch of 1000 claims, since similar work is done every day. When several equally important jobs are sitting in the input queue waiting to be started, the scheduler picks the **shortest job first**. Look at Fig. 2-39. Here we find four jobs *A*, *B*, *C*, and *D* with run times of 8, 4, 4, and 4 minutes, respectively. By running them in that order, the

turnaround time for A is 8 minutes, for B is 12 minutes, for C is 16 minutes, and for D is 20 minutes for an average of 14 minutes.

Figure 2-39. An example of shortest job first scheduling. (a) Running four jobs in the original order. (b) Running them in shortest job first order.

Now let us consider running these four jobs using shortest job first, as shown in Fig. 2-39(b). The turnaround times are now 4, 8, 12, and 20 minutes for an average of 11 minutes. Shortest job first is provably optimal. Consider the case of four jobs, with run times of a, b, c, and d, respectively. The first job finishes at time a, the second finishes at time $a + b$, and so on. The mean turnaround time is $(4a + 3b + 2c + d)/4$. It is clear that a contributes more to the average than the other times, so it should be the shortest job, with b next, then c, and finally d as the longest as it affects only its own turnaround time. The same argument applies equally well to any number of jobs.

It is worth pointing out that shortest job first is only optimal when all the jobs are available simultaneously. As a counterexample, consider five jobs, A through E, with run times of 2, 4, 1, 1, and 1, respectively. Their arrival times are 0, 0, 3, 3, and 3. Initially, only A or B can be chosen, since the other three jobs have not arrived yet. Using shortest job first we will run the jobs in the order A, B, C, D, E, for an average wait of 4.6. However, running them in the order B, C, D, E, A has an average wait of 4.4.

Shortest Remaining Time Next

A preemptive version of shortest job first is **shortest remaining time next**. With this algorithm, the scheduler always chooses the process whose remaining run time is the shortest. Again here, the run time has to be known in advance. When a new job arrives, its total time is compared to the current process' remaining time. If the new job needs less time to finish than the current process, the current process is suspended and the new job started. This scheme allows new short jobs to get good service.

Three-Level Scheduling

From a certain perspective, batch systems allow scheduling at three different levels, as illustrated in Fig. 2-40. As jobs arrive at the system, they are initially placed in an input queue stored on the disk. The **admission scheduler** decides

which jobs to admit to the system. The others are kept in the input queue until they are selected. A typical algorithm for admission control might be to look for a mix of compute-bound jobs and I/O-bound jobs. Alternatively, short jobs could be admitted quickly whereas longer jobs would have to wait. The admission scheduler is free to hold some jobs in the input queue and admit jobs that arrive later if it so chooses.

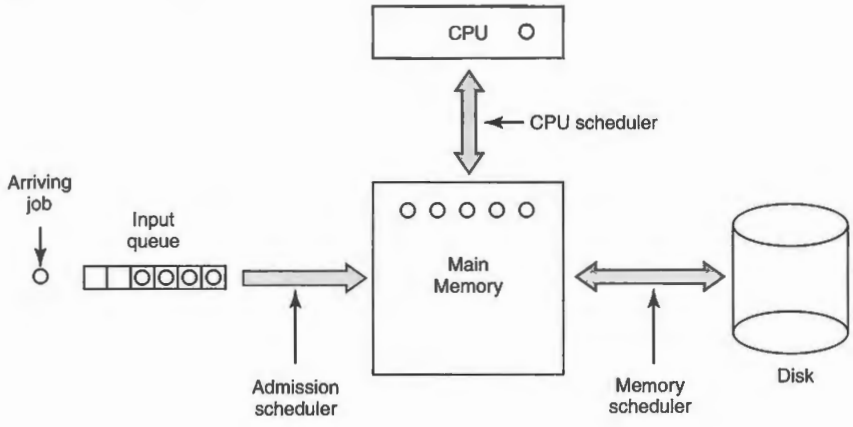


Figure 2-40. Three-level scheduling.

Once a job has been admitted to the system, a process can be created for it and it can contend for the CPU. However, it might well happen that the number of processes is so large that there is not enough room for all of them in memory. In that case, some of the processes have to be swapped out to disk. The second level of scheduling is deciding which processes should be kept in memory and which ones kept on disk. We will call this scheduler the **memory scheduler**, since it determines which processes are kept in memory and which on the disk.

This decision has to be reviewed frequently to allow the processes on disk to get some service. However, since bringing a process in from disk is expensive, the review probably should not happen more often than once per second, maybe less often. If the contents of main memory are shuffled too often, a large amount of disk bandwidth will be wasted, slowing down file I/O.

To optimize system performance as a whole, the memory scheduler might want to carefully decide how many processes it wants in memory, called the **degree of multiprogramming**, and what kind of processes. If it has information about which processes are compute bound and which are I/O bound, it can try to keep a mix of these process types in memory. As a very crude approximation, if a certain class of process computes about 20% of the time, keeping five of them around is roughly the right number to keep the CPU busy. We will look at a slightly better multiprogramming model in Chap. 4.

To make its decisions, the memory scheduler periodically reviews each process on disk to decide whether or not to bring it into memory. Among the criteria that it can use to make its decision are the following ones:

1. How long has it been since the process was swapped in or out?
2. How much CPU time has the process had recently?
3. How big is the process? (Small ones do not get in the way.)
4. How important is the process?

The third level of scheduling is actually picking one of the ready processes in main memory to run next. Often this is called the **CPU scheduler** and is the one people usually mean when they talk about the "scheduler." Any suitable algorithm can be used here, either preemptive or nonpreemptive. These include the ones described above as well as a number of algorithms to be described in the next section.

2.5.3 Scheduling in Interactive Systems

We will now look at some algorithms that can be used in interactive systems. All of these can also be used as the CPU scheduler in batch systems as well. While three-level scheduling is not possible here, two-level scheduling (memory scheduler and CPU scheduler) is possible and common. Below we will focus on the CPU scheduler.

Round-Robin Scheduling

Now let us look at some specific scheduling algorithms. One of the oldest, simplest, fairest, and most widely used algorithms is **round robin**. Each process is assigned a time interval, called its **quantum**, which it is allowed to run. If the process is still running at the end of the quantum, the CPU is preempted and given to another process. If the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks, of course. Round robin is easy to implement. All the scheduler needs to do is maintain a list of runnable processes, as shown in Fig. 2-41(a). When the process uses up its quantum, it is put on the end of the list, as shown in Fig. 2-41(b).

The only interesting issue with round robin is the length of the quantum. Switching from one process to another requires a certain amount of time for doing the administration—saving and loading registers and memory maps, updating various tables and lists, flushing and reloading the memory cache, etc. Suppose that this **process switch** or **context switch**, as it is sometimes called, takes 1 msec, including switching memory maps, flushing and reloading the cache, etc. Also suppose that the quantum is set at 4 msec. With these parameters, after doing 4

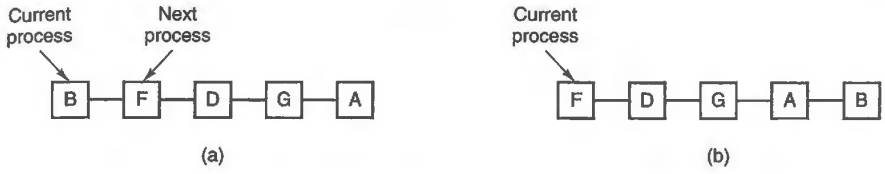


Figure 2-41. Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after *B* uses up its quantum.

msec of useful work, the CPU will have to spend 1 msec on process switching. Twenty percent of the CPU time will be wasted on administrative overhead. Clearly this is too much.

To improve the CPU efficiency, we could set the quantum to, say, 100 msec. Now the wasted time is only 1 percent. But consider what happens on a timesharing system if ten interactive users hit the carriage return key at roughly the same time. Ten processes will be put on the list of runnable processes. If the CPU is idle, the first one will start immediately, the second one may not start until 100 msec later, and so on. The unlucky last one may have to wait 1 sec before getting a chance, assuming all the others use their full quanta. Most users will perceive a 1-sec response to a short command as sluggish

Another factor is that if the quantum is set longer than the mean CPU burst, preemption will rarely happen. Instead, most processes will perform a blocking operation before the quantum runs out, causing a process switch. Eliminating preemption improves performance because process switches then only happen when they are logically necessary, that is, when a process blocks and cannot continue.

The conclusion can be formulated as follows: setting the quantum too short causes too many process switches and lowers the CPU efficiency, but setting it too long may cause poor response to short interactive requests. A quantum around 20-50 msec is often a reasonable compromise.

Priority Scheduling

Round robin scheduling makes the implicit assumption that all processes are equally important. Frequently, the people who own and operate multiuser computers have different ideas on that subject. At a university, the pecking order may be deans first, then professors, secretaries, janitors, and finally students. The need to take external factors into account leads to **priority scheduling**. The basic idea is straightforward: each process is assigned a priority, and the runnable process with the highest priority is allowed to run.

Even on a PC with a single owner, there may be multiple processes, some more important than others. For example, a daemon process sending electronic

mail in the background should be assigned a lower priority than a process displaying a video film on the screen in real time.

To prevent high-priority processes from running indefinitely, the scheduler may decrease the priority of the currently running process at each clock tick (i.e., at each clock interrupt). If this action causes its priority to drop below that of the next highest process, a process switch occurs. Alternatively, each process may be assigned a maximum time quantum that it is allowed to run. When this quantum is used up, the next highest priority process is given a chance to run.

Priorities can be assigned to processes statically or dynamically. On a military computer, processes started by generals might begin at priority 100, processes started by colonels at 90, majors at 80, captains at 70, lieutenants at 60, and so on. Alternatively, at a commercial computer center, high-priority jobs might cost 100 dollars an hour, medium priority 75 dollars an hour, and low priority 50 dollars an hour. The UNIX system has a command, *nice*, which allows a user to voluntarily reduce the priority of his process, in order to be nice to the other users. Nobody ever uses it.

Priorities can also be assigned dynamically by the system to achieve certain system goals. For example, some processes are highly I/O bound and spend most of their time waiting for I/O to complete. Whenever such a process wants the CPU, it should be given the CPU immediately, to let it start its next I/O request, which can then proceed in parallel with another process actually computing. Making the I/O-bound process wait a long time for the CPU will just mean having it around occupying memory for an unnecessarily long time. A simple algorithm for giving good service to I/O-bound processes is to set the priority to $1/f$, where f is the fraction of the last quantum that a process used. A process that used only 1 msec of its 50 msec quantum would get priority 50, while a process that ran 25 msec before blocking would get priority 2, and a process that used the whole quantum would get priority 1.

It is often convenient to group processes into priority classes and use priority scheduling among the classes but round-robin scheduling within each class. Figure 2-42 shows a system with four priority classes. The scheduling algorithm is as follows: as long as there are runnable processes in priority class 4, just run each one for one quantum, round-robin fashion, and never bother with lower priority classes. If priority class 4 is empty, then run the class 3 processes round robin. If classes 4 and 3 are both empty, then run class 2 round robin, and so on. If priorities are not adjusted occasionally, lower priority classes may all starve to death.

Multiple Queues

One of the earliest priority schedulers was in CTSS (Corbató et al., 1962). CTSS had the problem that process switching was very slow because the 7094 could hold only one process in memory. Each switch meant swapping the current process to disk and reading in a new one from disk. The CTSS designers quickly

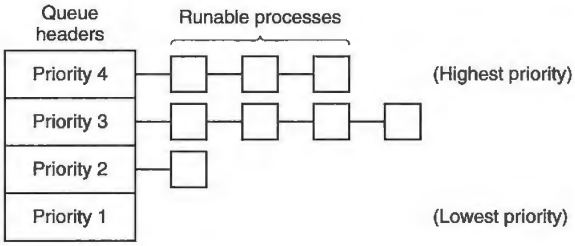


Figure 2-42. A scheduling algorithm with four priority classes.

realized that it was more efficient to give CPU-bound processes a large quantum once in a while, rather than giving them small quanta frequently (to reduce swapping). On the other hand, giving all processes a large quantum would mean poor response time, as we have already seen. Their solution was to set up priority classes. Processes in the highest class were run for one quantum. Processes in the next highest class were run for two quanta. Processes in the next class were run for four quanta, and so on. Whenever a process used up all the quanta allocated to it, it was moved down one class.

As an example, consider a process that needed to compute continuously for 100 quanta. It would initially be given one quantum, then swapped out. Next time it would get two quanta before being swapped out. On succeeding runs it would get 4, 8, 16, 32, and 64 quanta, although it would have used only 37 of the final 64 quanta to complete its work. Only 7 swaps would be needed (including the initial load) instead of 100 with a pure round-robin algorithm. Furthermore, as the process sank deeper and deeper into the priority queues, it would be run less and less frequently, saving the CPU for short, interactive processes.

The following policy was adopted to prevent a process that needed to run for a long time when it first started but became interactive later, from being punished forever. Whenever a carriage return was typed at a terminal, the process belonging to that terminal was moved to the highest priority class, on the assumption that it was about to become interactive. One fine day, some user with a heavily CPU-bound process discovered that just sitting at the terminal and typing carriage returns at random every few seconds did wonders for his response time. He told all his friends. Moral of the story: getting it right in practice is much harder than getting it right in principle.

Many other algorithms have been used for assigning processes to priority classes. For example, the influential XDS 940 system (Lampson, 1968), built at Berkeley, had four priority classes, called terminal, I/O, short quantum, and long quantum. When a process that was waiting for terminal input was finally awakened, it went into the highest priority class (terminal). When a process waiting for a disk block became ready, it went into the second class. When a process was still

running when its quantum ran out, it was initially placed in the third class. However, if a process used up its quantum too many times in a row without blocking for terminal or other I/O, it was moved down to the bottom queue. Many other systems use something similar to favor interactive users and processes over background ones.

Shortest Process Next

Because shortest job first always produces the minimum average response time for batch systems, it would be nice if it could be used for interactive processes as well. To a certain extent, it can be. Interactive processes generally follow the pattern of wait for command, execute command, wait for command, execute command, and so on. If we regard the execution of each command as a separate "job," then we could minimize overall response time by running the shortest one first. The only problem is figuring out which of the currently runnable processes is the shortest one.

One approach is to make estimates based on past behavior and run the process with the shortest estimated running time. Suppose that the estimated time per command for some terminal is T_0. Now suppose its next run is measured to be T_1. We could update our estimate by taking a weighted sum of these two numbers, that is, $aT_0 + (1 - a)T_1$. Through the choice of a we can decide to have the estimation process forget old runs quickly, or remember them for a long time. With $a = 1/2$, we get successive estimates of

$$T_0, \quad T_0/2 + T_1/2, \quad T_0/4 + T_1/4 + T_2/2, \quad T_0/8 + T_1/8 + T_2/4 + T_3/2$$

After three new runs, the weight of T_0 in the new estimate has dropped to 1/8.

The technique of estimating the next value in a series by taking the weighted average of the current measured value and the previous estimate is sometimes called **aging**. It is applicable to many situations where a prediction must be made based on previous values. Aging is especially easy to implement when $a = 1/2$. All that is needed is to add the new value to the current estimate and divide the sum by 2 (by shifting it right 1 bit).

Guaranteed Scheduling

A completely different approach to scheduling is to make real promises to the users about performance and then live up to them. One promise that is realistic to make and easy to live up to is this: If there are n users logged in while you are working, you will receive about $1/n$ of the CPU power. Similarly, on a single-user system with n processes running, all things being equal, each one should get $1/n$ of the CPU cycles.

To make good on this promise, the system must keep track of how much CPU each process has had since its creation. It then computes the amount of CPU each

one is entitled to, namely the time since creation divided by n. Since the amount of CPU time each process has actually had is also known, it is straightforward to compute the ratio of actual CPU time consumed to CPU time entitled. A ratio of 0.5 means that a process has only had half of what it should have had, and a ratio of 2.0 means that a process has had twice as much as it was entitled to. The algorithm is then to run the process with the lowest ratio until its ratio has moved above its closest competitor.

Lottery Scheduling

While making promises to the users and then living up to them is a fine idea, it is difficult to implement. However, another algorithm can be used to give similarly predictable results with a much simpler implementation. It is called **lottery scheduling** (Waldspurger and Weihl, 1994).

The basic idea is to give processes lottery tickets for various system resources, such as CPU time. Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource. When applied to CPU scheduling, the system might hold a lottery 50 times a second, with each winner getting 20 msec of CPU time as a prize.

To paraphrase George Orwell: "All processes are equal, but some processes are more equal." More important processes can be given extra tickets, to increase their odds of winning. If there are 100 tickets outstanding, and one process holds 20 of them, it will have a 20 percent chance of winning each lottery. In the long run, it will get about 20 percent of the CPU. In contrast to a priority scheduler, where it is very hard to state what having a priority of 40 actually means, here the rule is clear: a process holding a fraction f of the tickets will get about a fraction f of the resource in question.

Lottery scheduling has several interesting properties. For example, if a new process shows up and is granted some tickets, at the very next lottery it will have a chance of winning in proportion to the number of tickets it holds. In other words, lottery scheduling is highly responsive.

Cooperating processes may exchange tickets if they wish. For example, when a client process sends a message to a server process and then blocks, it may give all of its tickets to the server, to increase the chance of the server running next. When the server is finished, it returns the tickets so the client can run again. In fact, in the absence of clients, servers need no tickets at all.

Lottery scheduling can be used to solve problems that are difficult to handle with other methods. One example is a video server in which several processes are feeding video streams to their clients, but at different frame rates. Suppose that the processes need frames at 10, 20, and 25 frames/sec. By allocating these processes 10, 20, and 25 tickets, respectively, they will automatically divide the CPU in approximately the correct proportion, that is, 10 : 20 : 25.

Fair-Share Scheduling

So far we have assumed that each process is scheduled on its own, without regard to who its owner is. As a result, if user 1 starts up 9 processes and user 2 starts up 1 process, with round robin or equal priorities, user 1 will get 90% of the CPU and user 2 will get only 10% of it.

To prevent this situation, some systems take into account who owns a process before scheduling it. In this model, each user is allocated some fraction of the CPU and the scheduler picks processes in such a way as to enforce it. Thus if two users have each been promised 50% of the CPU, they will each get that, no matter how many processes they have in existence.

As an example, consider a system with two users, each of which has been promised 50% of the CPU. User 1 has four processes, *A*, *B*, *C*, and *D*, and user 2 has only 1 process, *E*. If round-robin scheduling is used, a possible scheduling sequence that meets all the constraints is this one:

A E B E C E D E A E B E C E D E ...

On the other hand, if user 1 is entitled to twice as much CPU time as user 2, we might get

A B E C D E A B E C D E ...

Numerous other possibilities exist, of course, and can be exploited, depending on what the notion of fairness is.

2.5.4 Scheduling in Real-Time Systems

A **real-time** system is one in which time plays an essential role. Typically, one or more physical devices external to the computer generate stimuli, and the computer must react appropriately to them within a fixed amount of time. For example, the computer in a compact disc player gets the bits as they come off the drive and must convert them into music within a very tight time interval. If the calculation takes too long, the music will sound peculiar. Other real-time systems are patient monitoring in a hospital intensive-care unit, the autopilot in an aircraft, and robot control in an automated factory. In all these cases, having the right answer but having it too late is often just as bad as not having it at all.

Real-time systems are generally categorized as **hard real time**, meaning there are absolute deadlines that must be met, or else, and **soft real time**, meaning that missing an occasional deadline is undesirable, but nevertheless tolerable. In both cases, real-time behavior is achieved by dividing the program into a number of processes, each of whose behavior is predictable and known in advance. These processes are generally short lived and can run to completion in well under a second. When an external event is detected, it is the job of the scheduler to schedule the processes in such a way that all deadlines are met.

The events that a real-time system may have to respond to can be further categorized as **periodic** (occurring at regular intervals) or **aperiodic** (occurring unpredictably). A system may have to respond to multiple periodic event streams. Depending on how much time each event requires for processing, it may not even be possible to handle them all. For example, if there are m periodic events and event i occurs with period P_i and requires C_i seconds of CPU time to handle each event, then the load can only be handled if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

A real-time system that meets this criteria is said to be **schedulable**.

As an example, consider a soft real-time system with three periodic events, with periods of 100, 200, and 500 msec, respectively. If these events require 50, 30, and 100 msec of CPU time per event, respectively, the system is schedulable because $0.5 + 0.15 + 0.2 < 1$. If a fourth event with a period of 1 sec is added, the system will remain schedulable as long as this event does not need more than 150 msec of CPU time per event. Implicit in this calculation is the assumption that the context-switching overhead is so small that it can be ignored.

Real-time scheduling algorithms can be static or dynamic. The former make their scheduling decisions before the system starts running. The latter make their scheduling decisions at run time. Static scheduling only works when there is perfect information available in advance about the work needed to be done and the deadlines that have to be met. Dynamic scheduling algorithms do not have these restrictions. We will defer our study of specific algorithms until we treat real-time multimedia systems in Chap. 7.

2.5.5 Policy versus Mechanism

Up until now, we have tacitly assumed that all the processes in the system belong to different users and are thus competing for the CPU. While this is often true, sometimes it happens that one process has many children running under its control. For example, a database management system process may have many children. Each child might be working on a different request, or each one might have some specific function to perform (query parsing, disk access, etc.). It is entirely possible that the main process has an excellent idea of which of its children are the most important (or time critical) and which the least. Unfortunately, none of the schedulers discussed above accept any input from user processes about scheduling decisions. As a result, the scheduler rarely makes the best choice.

The solution to this problem is to separate the **scheduling mechanism** from the **scheduling policy**. What this means is that the scheduling algorithm is parameterized in some way, but the parameters can be filled in by user processes. Let us consider the database example once again. Suppose that the kernel uses a

priority scheduling algorithm but provides a system call by which a process can set (and change) the priorities of its children. In this way the parent can control in detail how its children are scheduled, even though it itself does not do the scheduling. Here the mechanism is in the kernel but policy is set by a user process.

2.5.6 Thread Scheduling

When several processes each have multiple threads, we have two levels of parallelism present: processes and threads. Scheduling in such systems differs substantially depending on whether user-level threads or kernel-level threads (or both) are supported.

Let us consider user-level threads first. Since the kernel is not aware of the existence of threads, it operates as it always does, picking a process, say, *A*, and giving *A* control for its quantum. The thread scheduler inside *A* decides which thread to run, say *A1*. Since there are no clock interrupts to multiprogram threads, this thread may continue running as long as it wants to. If it uses up the process' entire quantum, the kernel will select another process to run.

When the process *A* finally runs again, thread *A1* will resume running. It will continue to consume all of *A*'s time until it is finished. However, its antisocial behavior will not affect other processes. They will get whatever the scheduler considers their appropriate share, no matter what is going on inside process *A*.

Now consider the case that *A*'s threads have relatively little work to do per CPU burst, for example, 5 msec of work within a 50-msec quantum. Consequently, each one runs for a little while, then yields the CPU back to the thread scheduler. This might lead to the sequence *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, before the kernel switches to process *B*. This situation is illustrated in Fig. 2-43(a).

The scheduling algorithm used by the run-time system can be any of the ones described above. In practice, round-robin scheduling and priority scheduling are most common. The only constraint is the absence of a clock to interrupt a thread that has run too long.

Now consider the situation with kernel-level threads. Here the kernel picks a particular thread to run. It does not have to take into account which process the thread belongs to, but it can if it wants to. The thread is given a quantum and is forceably suspended if it exceeds the quantum. With a 50-msec quantum but threads that block after 5 msec, the thread order for some period of 30 msec might be *A1*, *B1*, *A2*, *B2*, *A3*, *B3*, something not possible with these parameters and user-level threads. This situation is partially depicted in Fig. 2-43(b).

A major difference between user-level threads and kernel-level threads is the performance. Doing a thread switch with user-level threads takes a handful of machine instructions. With kernel-level threads it requires a full context switch, changing the memory map, and invalidating the cache, which is several orders of

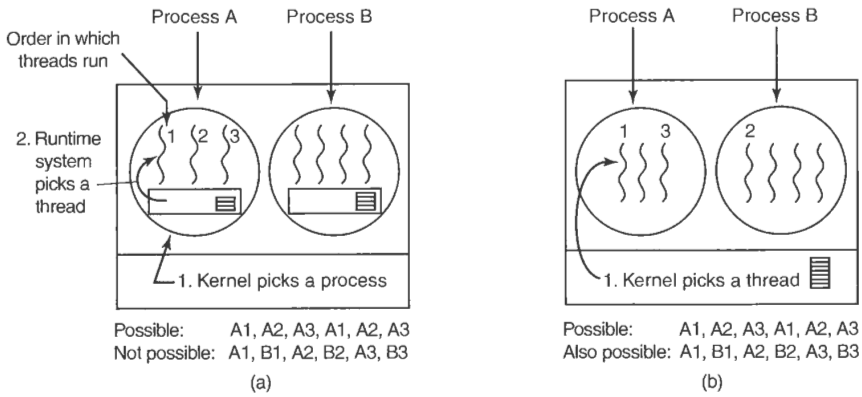


Figure 2-43. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

magnitude slower. On the other hand, with kernel-level threads, having a thread block on I/O does not suspend the entire process as it does with user-level threads.

Since the kernel knows that switching from a thread in process *A* to a thread in process *B* is more expensive that running a second thread in process *A* (due to having to change the memory map and having the memory cache spoiled), it can take this information into account when making a decision. For example, given two threads that are otherwise equally important, with one of them belonging to the same process as a thread that just blocked and one belonging to a different process, preference could be given to the former.

Another important factor is that user-level threads can employ an application-specific thread scheduler. Consider, for example, the Web server of Fig. 2-10. Suppose that a worker thread has just blocked and the dispatcher thread and two worker threads are ready. Who should run next? The run-time system, knowing what all the threads do, can easily pick the dispatcher to run next, so it can start another worker running. This strategy maximizes the amount of parallelism in an environment where workers frequently block on disk I/O. With kernel-level threads, the kernel would never know what each thread did (although they could be assigned different priorities). In general, however, application-specific thread schedulers can tune an application better than the kernel can.

2.6 RESEARCH ON PROCESSES AND THREADS

In Chap. 1, we looked at some of the current research in operating system structure. In this and subsequent chapters we will look at more narrowly focused research, starting with processes. As will become clear in time, some subjects are

much more settled than others. Most of the research tends to be on the new topics, rather than ones that have been around for decades.

The concept of a process is an example of something that is well settled. Almost every system has some notion of a process as a container for grouping together related resources such as an address space, threads, open files, protection permissions, etc. Different systems do the grouping slightly differently, but these are just engineering differences. The basic idea is not very controversial any more and there is little new research on the subject.

Threads are a newer idea than processes, so there is still some research going on about them. Hauser et al. (1993) looked at how real programs actually use threads and came up with 10 different paradigms for thread usage. Thread scheduling (both uniprocessor and multiprocessor) is still a topic near and dear to the heart of some researchers (Blumofe and Leiserson, 1994; Buchanan and Chien, 1997; Corbalán et al., 2000; Chandra et al., 2000; Duda and Cheriton, 1999; Ford and Susarla, 1996; and Petrou et al., 1999). However, few actual system designers are walking around all day wringing their hands for lack of a decent thread scheduling algorithm, so it appears this type of research is more researcher-push than demand-pull.

Closely related to threads is thread synchronization and mutual exclusion. In the 1970s and 1980s, that subject was mined for all it was worth, so there is not much current work on the subject, and what there is tends to be focuses on performance (e.g., Liedtke, 1993) tools for detecting synchronization errors (Savage et al., 1997), or modifying old concepts in new ways (Tai and Carver, 1996; Trono, 2000). Finally, new POSIX-conformant threads packages are still being produced and reported on (Alfieri, 1994; and Miller, 1999).

2.7 SUMMARY

To hide the effects of interrupts, operating systems provide a conceptual model consisting of sequential processes running in parallel. Processes can be created and terminated dynamically. Each process has its own address space.

For some applications it is useful to have multiple threads of control within a single process. These threads are scheduled independently and each one has its own stack, but all the threads in a process share a common address space. Threads can be implemented in user space or in the kernel.

Processes can communicate with each other using interprocess communication primitives, such as semaphores, monitors, or messages. These primitives are used to ensure that no two processes are ever in their critical regions at the same time, a situation that leads to chaos. A process can be running, runnable, or blocked and can change state when it or another process executes one of the interprocess communication primitives. Interthread communication is similar.

Interprocess communication primitives can be used to solve such problems as the producer-consumer, dining philosophers, reader-writer, and sleeping barber. Even with these primitives, care has to be taken to avoid errors and deadlocks.

Many scheduling algorithms are known. Some of these are primarily used for batch systems, such as shortest job first. Others are common in both batch systems and interactive systems. These include round robin, priority scheduling, multilevel queues, guaranteed scheduling, lottery scheduling, and fair-share scheduling. Some systems make a clean separation between the scheduling mechanism and the scheduling policy, which allows users to have control of the scheduling algorithm.

PROBLEMS

1. In Fig. 2-2, three process states are shown. In theory, with three states, there could be six transitions, two out of each state. However, only four transitions are shown. Are there any circumstances in which either or both of the missing transitions might occur?
2. Suppose that you were to design an advanced computer architecture that did process switching in hardware, instead of having interrupts. What information would the CPU need? Describe how the hardware process switching might work.
3. On all current computers, at least part of the interrupt handlers are written in assembly language. Why?
4. When an interrupt or a system call transfers control to the operating system, a kernel stack area separate from the stack of the interrupted process is generally used. Why?
5. In the text it was stated that the model of Fig. 2-6(a) was not suited to a file server using a cache in memory. Why not? Could each process have its own cache?
6. In Fig. 2-7 the register set is listed as a per-thread rather than a per-process item. Why? After all, the machine has only one set of registers.
7. If a multithreaded process forks, a problem occurs if the child gets copies of all the parent's threads. Suppose that one of the original threads was waiting for keyboard input. Now two threads are waiting for keyboard input, one in each process. Does this problem ever occur in single-threaded processes?
8. In Fig. 2-10, a multithreaded Web server is shown. If the only way to read from a file is the normal blocking read system call, do you think user-level threads or kernel-level threads are being used for the Web server? Why?
9. Why would a thread ever voluntarily give up the CPU by calling *thread_yield*? After all, since there is no periodic clock interrupt, it may never get the CPU back.
10. Can a thread ever be preempted by a clock interrupt? If so, under what circumstances? If not, why not?

11. In this problem you are to compare reading a file using a single-threaded file server and a multithreaded server. It takes 15 msec to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in the block cache. If a disk operation is needed, as is the case one-third of the time, an additional 75 msec is required, during which time the thread sleeps. How many requests/sec can the server handle if it is single threaded? If it is multithreaded?

12. In the text, we described a multithreaded Web server, showing why it is better than a single-threaded server and a finite-state machine server. Are there any circumstances in which a single-threaded server might be better? Give an example.

13. In the discussion on global variables in threads, we used a procedure *create_global* to allocate storage for a pointer to the variable, rather than the variable itself. Is this essential, or could the procedures work with the values themselves just as well?

14. Consider a system in which threads are implemented entirely in user space, with the run-time system getting a clock interrupt once a second. Suppose that a clock interrupt occurs while some thread is executing in the run-time system. What problem might occur? Can you suggest a way to solve it?

15. Suppose that an operating system does not have anything like the select system call to see in advance if it is safe to read from a file, pipe, or device, but it does allow alarm clocks to be set that interrupt blocked system calls. Is it possible to implement a threads package in user space under these conditions? Discuss.

16. Can the priority inversion problem discussed in Sec. 2.3.4 happen with user-level threads? Why or why not?

17. In a system with threads, is there one stack per thread or one stack per process when user-level threads are used? What about when kernel-level threads are used? Explain.

18. What is a race condition?

19. When a computer is being developed, it is usually first simulated by a program that runs one instruction at a time. Even multiprocessors are simulated strictly sequentially like this. Is it possible for a race condition to occur when there are no simultaneous events like this?

20. Does the busy waiting solution using the *turn* variable (Fig. 2-20) work when the two processes are running on a shared-memory multiprocessor, that is, two CPUs, sharing a common memory?

21. Does Peterson's solution to the mutual exclusion problem shown in Fig. 2-21 work when process scheduling is preemptive? How about when it is nonpreemptive?

22. Consider a computer that does not have a TSL instruction but does have an instruction to swap the contents of a register and a memory word in a single indivisible action. Can that be used to write a routine *enter_region* such as the one found in Fig. 2-22?

23. Give a sketch of how an operating system that can disable interrupts could implement semaphores.

24. Show how counting semaphores (i.e., semaphores that can hold an arbitrary value) can be implemented using only binary semaphores and ordinary machine instructions.

25. If a system has only two processes, does it make sense to use a barrier to synchronize them? Why or why not?

26. In Sec. 2.3.4, a situation with a high-priority process, *H*, and a low-priority process, *L*, was described, which led to *H* looping forever. Does the same problem occur if round-robin scheduling is used instead of priority scheduling? Discuss.

27. Can two threads in the same process synchronize using a kernel semaphore if the threads are implemented by the kernel? What if they are implemented in user space? Assume that no threads in any other processes have access to the semaphore. Discuss your answers.

28. Synchronization within monitors uses condition variables and two special operations, wait and signal. A more general form of synchronization would be to have a single primitive, waituntil, that had an arbitrary Boolean predicate as parameter. Thus, one could say, for example,

waituntil $x < 0$ or $y + z < n$

The signal primitive would no longer be needed. This scheme is clearly more general than that of Hoare or Brinch Hansen, but it is not used. Why not? *Hint*: Think about the implementation.

29. A fast food restaurant has four kinds of employees: (1) order takers, who take customers' orders; (2) cooks, who prepare the food; (3) packaging specialists, who stuff the food into bags; and (4) cashiers, who give the bags to customers and take their money. Each employee can be regarded as a communicating sequential process. What form of interprocess communication do they use? Relate this model to processes in UNIX.

30. Suppose that we have a message-passing system using mailboxes. When sending to a full mailbox or trying to receive from an empty one, a process does not block. Instead, it gets an error code back. The process responds to the error code by just trying again, over and over, until it succeeds. Does this scheme lead to race conditions?

31. In the solution to the dining philosophers problem (Fig. 2-20), why is the state variable set to *HUNGRY* in the procedure *take_forks*?

32. Consider the procedure *put_forks* in Fig. 2-20. Suppose that the variable *state*[*i*] was set to *THINKING after* the two calls to *test*, rather than *before*. How would this change affect the solution?

33. The readers and writers problem can be formulated in several ways with regard to which category of processes can be started when. Carefully describe three different variations of the problem, each one favoring (or not favoring) some category of processes. For each variation, specify what happens when a reader or a writer becomes ready to access the database, and what happens when a process is finished using the database.

34. The CDC 6600 computers could handle up to 10 I/O processes simultaneously using an interesting form of round-robin scheduling called **processor sharing**. A process switch occurred after each instruction, so instruction 1 came from process 1, instruction 2 came from process 2, etc. The process switching was done by special hardware, and the overhead was zero. If a process needed *T* sec to complete in the absence of

competition, how much time would it need if processor sharing was used with n processes?

35. Round-robin schedulers normally maintain a list of all runnable processes, with each process occurring exactly once in the list. What would happen if a process occurred twice in the list? Can you think of any reason for allowing this?

36. Can a measure of whether a process is likely to be CPU bound or I/O bound be determined by analyzing source code? How can this be determined at run time?

37. In the section "When to Schedule," it was mentioned that sometimes scheduling could be improved if an important process could play a role in selecting the next process to run when it blocks. Give a situation where this could be used and explain how.

38. Measurements of a certain system have shown that the average process runs for a time T before blocking on I/O. A process switch requires a time S, which is effectively wasted (overhead). For round-robin scheduling with quantum Q, give a formula for the CPU efficiency for each of the following:

(a) $Q = \infty$
(b) $Q > T$
(c) $S < Q < T$
(d) $Q = S$
(e) Q nearly 0

39. Five jobs are waiting to be run. Their expected run times are 9, 6, 3, 5, and X. In what order should they be run to minimize average response time? (Your answer will depend on X.)

40. Five batch jobs A through E, arrive at a computer center at almost the same time. They have estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally determined) priorities are 3, 5, 2, 1, and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the mean process turnaround time. Ignore process switching overhead.

(a) Round robin.
(b) Priority scheduling.
(c) First-come, first-served (run in order 10, 6, 2, 4, 8).
(d) Shortest job first.

For (a), assume that the system is multiprogrammed, and that each job gets its fair share of the CPU. For (b) through (d) assume that only one job at a time runs, until it finishes. All jobs are completely CPU bound.

41. A process running on CTSS needs 30 quanta to complete. How many times must it be swapped in, including the very first time (before it has run at all)?

42. Can you think of a way to save the CTSS priority system from being fooled by random carriage returns?

43. The aging algorithm with $a = 1/2$ is being used to predict run times. The previous four runs, from oldest to most recent, are 40, 20, 40, and 15 msec. What is the prediction of the next time?

44. A soft real-time system has four periodic events with periods of 50, 100, 200, and 250 msec each. Suppose that the four events require 35, 20, 10, and x msec of CPU time, respectively. What is the largest value of x for which the system is schedulable?

45. Explain why two-level scheduling is commonly used.

46. Consider a system in which it is desired to separate policy and mechanism for the scheduling of kernel threads. Propose a means of achieving this goal.

47. Write a shell script that produces a file of sequential numbers by reading the last number in the file, adding 1 to it, and then appending it to the file. Run one instance of the script in the background and one in the foreground, each accessing the same file. How long does it take before a race condition manifests itself? What is the critical region? Modify the script to prevent the race (hint: use

```
ln file file.lock
```

to lock the data file).

48. Assume that you have an operating system that provides semaphores. Implement a message system. Write the procedures for sending and receiving messages.

49. Solve the dining philosophers problem using monitors instead of semaphores.

50. Suppose that a university wants to show off how politically correct it is by applying the U.S. Supreme Court's "Separate but equal is inherently unequal" doctrine to gender as well as race, ending its long-standing practice of gender-segregated bathrooms on campus. However, as a concession to tradition, it decrees that when a woman is in a bathroom, other women may enter, but no men, and vice versa. A sign with a sliding marker on the door of each bathroom indicates which of three possible states it is currently in:

- Empty
- Women present
- Men present

In your favorite programming language, write the following procedures: *woman_wants_to_enter*, *man_wants_to_enter*, *woman_leaves*, *man_leaves*. You may use whatever counters and synchronization techniques you like.

51. Rewrite the program of Fig. 2-20 to handle more than two processes.

52. Write a producer-consumer problem that uses threads and shares a common buffer. However, do not use semaphores or any other synchronization primitives to guard the shared data structures. Just let each thread access them when it wants to. Use sleep and wakeup to handle the full and empty conditions. See how long it takes for a fatal race condition to occur. For example, you might have the producer print a number once in a while. Do not print more than one number every minute because the I/O could affect the race conditions.

53. A process can be put into a round-robin queue more than once to give it a higher priority. Running multiple instances of a program each working on a different part of a data pool can have the same effect. First write a program that tests a list of numbers for primality. Then devise a method to allow multiple instances of the program to run

at once in such a way that no two instances of the program will work on the same number. Can you in fact get through the list faster by running multiple copies of the program? Note that your results will depend upon what else your computer is doing; on a personal computer running only instances of this program you would not expect an improvement, but on a system with other processes, you should be able to grab a bigger share of the CPU this way.

3

DEADLOCKS

Computer systems are full of resources that can only be used by one process at a time. Common examples include printers, tape drives, and slots in the system's internal tables. Having two processes simultaneously writing to the printer leads to gibberish. Having two processes using the same file system table slot will invariably lead to a corrupted file system. Consequently, all operating systems have the ability to (temporarily) grant a process exclusive access to certain resources.

For many applications, a process needs exclusive access to not one resource, but several. Suppose, for example, two processes each want to record a scanned document on a CD. Process *A* requests permission to use the scanner and is granted it. Process *B* is programmed differently and requests the CD recorder first and is also granted it. Now *A* asks for the CD recorder, but the request is denied until *B* releases it. Unfortunately, instead of releasing the CD recorder *B* asks for the scanner. At this point both processes are blocked and will remain so forever. This situation is called a **deadlock**.

Deadlocks can also occur across machines. For example, many offices have a local area network with many computers connected to it. Often devices such as scanners, CD recorders, printers, and tape drives are connected to the network as shared resources, available to any user on any machine. If these devices can be reserved remotely (i.e., from the user's home machine), the same kind of deadlocks can occur as described above. More complicated situations can cause deadlocks involving three, four, or more devices and users.

IBM Ex-1011
Page 183

Deadlocks can occur in a variety of situations besides requesting dedicated I/O devices. In a database system, for example, a program may have to lock several records it is using, to avoid race conditions. If process *A* locks record *R1* and process *B* locks record *R2*, and then each process tries to lock the other one's record, we also have a deadlock. Thus deadlocks can occur on hardware resources or on software resources.

In this chapter, we will look at deadlocks more closely, see how they arise, and study some ways of preventing or avoiding them. Although this material is about deadlocks in the context of operating systems, they also occur in database systems and many other contexts in computer science, so this material is actually applicable to a wide variety of multiprocess systems. A great deal has been written about deadlocks. Two bibliographies on the subject have appeared in *Operating Systems Review* and should be consulted for references (Newton, 1979; and Zobel, 1983). Although these bibliographies are old, most of the work on deadlocks was done well before 1980, so they are still useful.

3.1 RESOURCES

Deadlocks can occur when processes have been granted exclusive access to devices, files, and so forth. To make the discussion of deadlocks as general as possible, we will refer to the objects granted as **resources**. A resource can be a hardware device (e.g., a tape drive) or a piece of information (e.g., a locked record in a database). A computer will normally have many different resources that can be acquired. For some resources, several identical instances may be available, such as three tape drives. When several copies of a resource are available, any one of them can be used to satisfy any request for the resource. In short, a resource is anything that can be used by only a single process at any instant of time.

3.1.1 Preemptable and Nonpreemptable Resources

Resources come in two types: preemptable and nonpreemptable. A **preemptable resource** is one that can be taken away from the process owning it with no ill effects. Memory is an example of a preemptable resource. Consider, for example, a system with 32 MB of user memory, one printer, and two 32-MB processes that each want to print something. Process *A* requests and gets the printer, then starts to compute the values to print. Before it has finished with the computation, it exceeds its time quantum and is swapped out.

Process *B* now runs and tries, unsuccessfully, to acquire the printer. Potentially, we now have a deadlock situation, because *A* has the printer and *B* has the

memory, and neither can proceed without the resource held by the other. Fortunately, it is possible to preempt (take away) the memory from *B* by swapping it out and swapping *A* in. Now *A* can run, do its printing, and then release the printer. No deadlock occurs.

A **nonpreemptable resource**, in contrast, is one that cannot be taken away from its current owner without causing the computation to fail. If a process has begun to burn a CD-ROM, suddenly taking the CD recorder away from it and giving it to another process will result in a garbled CD. CD recorders are not preemptable at an arbitrary moment.

In general, deadlocks involve nonpreemptable resources. Potential deadlocks that involve preemptable resources can usually be resolved by reallocating resources from one process to another. Thus our treatment will focus on nonpreemptable resources.

The sequence of events required to use a resource is given below in an abstract form.

1. Request the resource.
2. Use the resource.
3. Release the resource.

If the resource is not available when it is requested, the requesting process is forced to wait. In some operating systems, the process is automatically blocked when a resource request fails, and awakened when it becomes available. In other systems, the request fails with an error code, and it is up to the calling process to wait a little while and try again.

A process whose resource request has just been denied will normally sit in a tight loop requesting the resource, then sleeping, then trying again. Although this process is not blocked, for all intents and purposes, it is as good as blocked, because it cannot do any useful work. In our further treatment, we will assume that when a process is denied a resource request, it is put to sleep.

The exact nature of requesting a resource is highly system dependent. In some systems, a request system call is provided to allow processes to explicitly ask for resources. In others, the only resources that the operating system knows about are special files that only one process can have open at a time. These are opened by the usual open call. If the file is already in use, the caller is blocked until its current owner closes it.

3.1.2 Resource Acquisition

For some kinds of resources, such as records in a database system, it is up to the user processes to manage resource usage themselves. One possible way of allowing user management of resources is to associate a semaphore with each resource. These semaphores are all initialized to 1. Mutexes can be used equally

well. The three steps listed above are then implemented as a down on the semaphore to acquire the resource, using the resource, and finally an up on the resource to release it. These steps are shown in Fig. 3-1(a).

```
typedef int semaphore;
semaphore resource_1;

void process_A(void) {
    down(&resource_1);
    use_resource_1( );
    up(&resource_1);
}
```

(a)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}
```

(b)

Figure 3-1. Using a semaphore to protect resources. (a) One resource. (b) Two resources.

Sometimes processes need two or more resources. They can be acquired sequentially, as shown in Fig. 3-1(b). If more than two resources are needed, they are just acquired one after another.

So far, so good. As long as only one process is involved, everything works fine. Of course, with only one process, there is no need to formally acquire resources, since there is no competition for them.

Now let us consider a situation with two processes, *A* and *B*, and two resources. Two scenarios are depicted in Fig. 3-2. In Fig. 3-2(a), both processes ask for the resources in the same order. In Fig. 3-2(b), they ask for them in a different order. This difference may seem minor, but it is not.

In Fig. 3-2(a), one of the processes will acquire the first resource before the other one. That process will then successfully acquire the second resource and do its work. If the other process attempts to acquire resource 1 before it has been released, the other process will simply block until it becomes available.

In Fig. 3-2(b), the situation is different. It might happen that one of the processes acquires both resources and effectively blocks out the other process until it is done. However, it might also happen that process *A* acquires resource 1 and process *B* acquires resource 2. Each one will now block when trying the acquire the other one. Neither process will ever run again. This situation is a deadlock.

Here we see how what appears to be a minor difference in coding style—which resource to acquire first—turns out to make the difference between the program working and the program failing in a hard-to-detect way. Because deadlocks can occur so easily, a lot of research has gone into ways to deal them. This chapter discusses deadlocks in detail and what can be done about them.

```
typedef int semaphore;
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }
```

(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}
```

(b)

Figure 3-2. (a) Deadlock-free code. (b) Code with a potential deadlock.

3.2 INTRODUCTION TO DEADLOCKS

Deadlock can be defined formally as follows:

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

Because all the processes are waiting, none of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever. For this model, we assume that processes have only a single thread and that there are no interrupts possible to wake up a blocked process. The no-interrupts condition is needed to prevent an otherwise deadlocked process from being awakened by, say, an alarm, and then causing events that release other processes in the set.

In most cases, the event that each process is waiting for is the release of some resource currently possessed by another member of the set. In other words, each member of the set of deadlocked processes is waiting for a resource that is owned by a deadlocked process. None of the processes can run, none of them can release any resources, and none of them can be awakened. The number of processes and the number and kind of resources possessed and requested are unimportant. This result holds for any kind of resource, including both hardware and software.

3.2.1 Conditions for Deadlock

Coffman et al. (1971) showed that four conditions must hold for there to be a deadlock:

1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.
2. Hold and wait condition. Processes currently holding resources granted earlier can request new resources.
3. No preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. Circular wait condition. There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

All four of these conditions must be present for a deadlock to occur. If one of them is absent, no deadlock is possible.

It is worth noting that each condition relates to a policy that a system can have or not have. Can a given resource be assigned to more than one process at once? Can a process hold a resource and ask for another? Can resources be preempted? Can circular waits exist? Later on we will see how deadlocks can be attacked by trying to negate some of these conditions.

3.2.2 Deadlock Modeling

Holt (1972) showed how these four conditions can be modeled using directed graphs. The graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares. An arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process. In Fig. 3-3(a), resource *R* is currently assigned to process *A*.

An arc from a process to a resource means that the process is currently blocked waiting for that resource. In Fig. 3-3(b), process *B* is waiting for resource *S*. In Fig. 3-3(c) we see a deadlock: process *C* is waiting for resource *T*, which is currently held by process *D*. Process *D* is not about to release resource *T* because it is waiting for resource *U*, held by *C*. Both processes will wait forever. A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle (assuming that there is one resource of each kind). In this example, the cycle is *C–T–D–U–C*.

Now let us look at an example of how resource graphs can be used. Imagine that we have three processes, *A*, *B*, and *C*, and three resources, *R*, *S*, and *T*. The

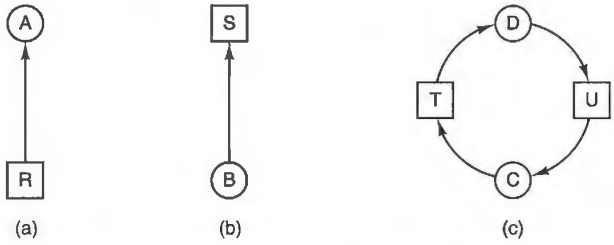


Figure 3-3. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

requests and releases of the three processes are given in Fig. 3-4(a)-(c). The operating system is free to run any unblocked process at any instant, so it could decide to run *A* until *A* finished all its work, then run *B* to completion, and finally run *C*.

This ordering does not lead to any deadlocks (because there is no competition for resources) but it also has no parallelism at all. In addition to requesting and releasing resources, processes compute and do I/O. When the processes are run sequentially, there is no possibility that while one process is waiting for I/O, another can use the CPU. Thus running the processes strictly sequentially may not be optimal. On the other hand, if none of the processes do any I/O at all, shortest job first is better than round robin, so under some circumstances running all processes sequentially may be the best way.

Let us now suppose that the processes do both I/O and computing, so that round robin is a reasonable scheduling algorithm. The resource requests might occur in the order of Fig. 3-4(d). If these six requests are carried out in that order, the six resulting resource graphs are shown in Fig. 3-4(e)-(j). After request 4 has been made, *A* blocks waiting for *S*, as shown in Fig. 3-4(h). In the next two steps *B* and *C* also block, ultimately leading to a cycle and the deadlock of Fig. 3-4(j).

However, as we have already mentioned, the operating system is not required to run the processes in any special order. In particular, if granting a particular request might lead to deadlock, the operating system can simply suspend the process without granting the request (i.e., just not schedule the process) until it is safe. In Fig. 3-4, if the operating system knew about the impending deadlock, it could suspend *B* instead of granting it *S*. By running only *A* and *C*, we would get the requests and releases of Fig. 3-4(k) instead of Fig. 3-4(d). This sequence leads to the resource graphs of Fig. 3-4(l)-(q), which do not lead to deadlock.

After step (q), process *B* can be granted *S* because *A* is finished and *C* has everything it needs. Even if *B* should eventually block when requesting *T*, no deadlock can occur. *B* will just wait until *C* is finished.

Later in this chapter we will study a detailed algorithm for making allocation decisions that do not lead to deadlock. For the moment, the point to understand is

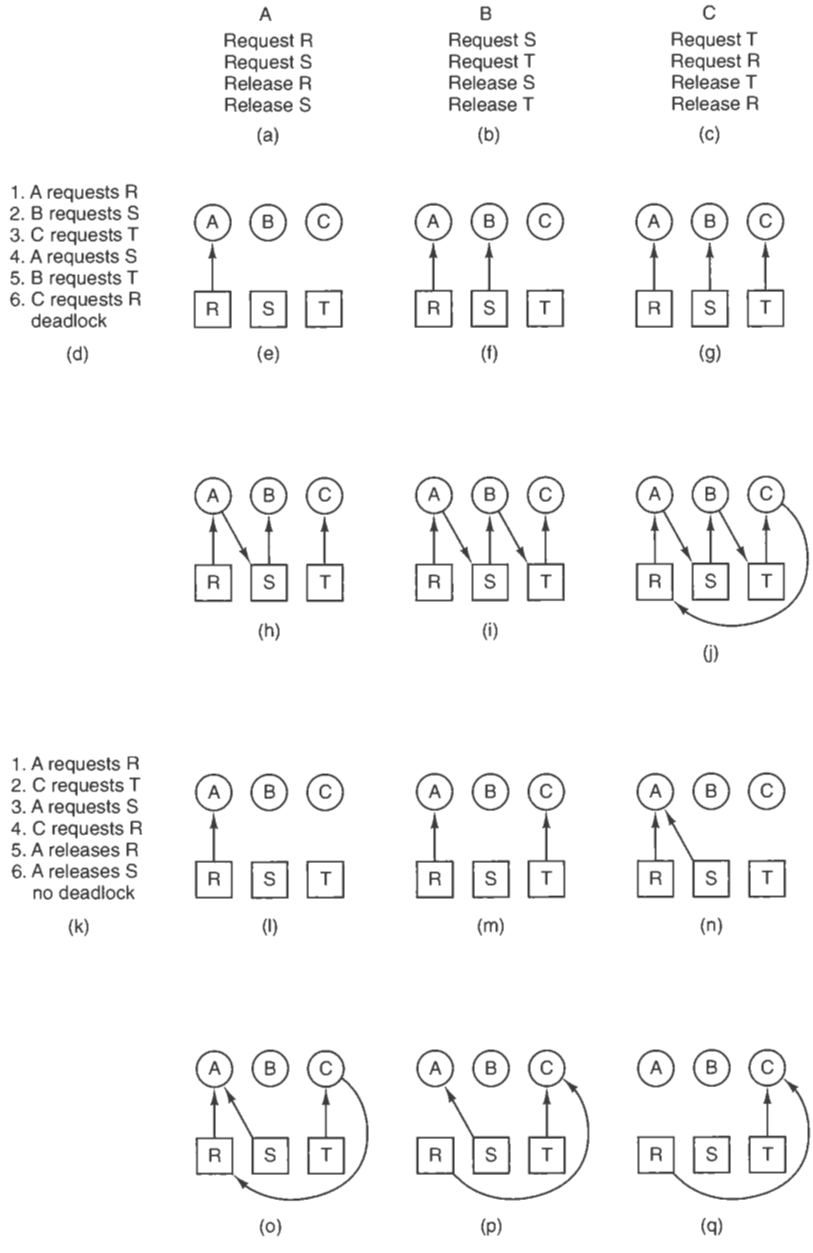


Figure 3-4. An example of how deadlock occurs and how it can be avoided.

that resource graphs are a tool that let us see if a given request/release sequence leads to deadlock. We just carry out the requests and releases step by step, and after every step check the graph to see if it contains any cycles. If so, we have a deadlock; if not, there is no deadlock. Although our treatment of resource graphs has been for the case of a single resource of each type, resource graphs can also be generalized to handle multiple resources of the same type (Holt, 1972).

In general, four strategies are used for dealing with deadlocks.

1. Just ignore the problem altogether. Maybe if you ignore it, it will ignore you.
2. Detection and recovery. Let deadlocks occur, detect them, and take action.
3. Dynamic avoidance by careful resource allocation.
4. Prevention, by structurally negating one of the four conditions necessary to cause a deadlock.

We will examine each of these methods in turn in the next four sections.

3.3 THE OSTRICH ALGORITHM

The simplest approach is the ostrich algorithm: stick your head in the sand and pretend there is no problem at all†. Different people react to this strategy in different ways. Mathematicians find it totally unacceptable and say that deadlocks must be prevented at all costs. Engineers ask how often the problem is expected, how often the system crashes for other reasons, and how serious a deadlock is. If deadlocks occur on the average once every five years, but system crashes due to hardware failures, compiler errors, and operating system bugs occur once a week, most engineers would not be willing to pay a large penalty in performance or convenience to eliminate deadlocks.

To make this contrast more specific, most operating systems potentially suffer from deadlocks that are not even detected, let alone automatically broken. The total number of processes in a system is determined by the number of entries in the process table. Thus process table slots are finite resources. If a fork fails because the table is full, a reasonable approach for the program doing the fork is to wait a random time and try again.

Now suppose that a UNIX system has 100 process slots. Ten programs are running, each of which needs to create 12 (sub)processes. After each process has

†Actually, this bit of folklore is nonsense. Ostriches can run at 60 km/hour and their kick is powerful enough to kill any lion with visions of a big chicken dinner.

created 9 processes, the 10 original processes and the 90 new processes have exhausted the table. Each of the 10 original processes now sits in an endless loop forking and failing—a deadlock. The probability of this happening is minuscule, but it *could* happen. Should we abandon processes and the fork call to eliminate the problem?

The maximum number of open files is similarly restricted by the size of the i-node table, so a similar problem occurs when it fills up. Swap space on the disk is another limited resource. In fact, almost every table in the operating system represents a finite resource. Should we abolish all of these because it might happen that a collection of n processes might each claim $1/n$ of the total, and then each try to claim another one?

Most operating systems, including UNIX and Windows, just ignore the problem on the assumption that most users would prefer an occasional deadlock to a rule restricting all users to one process, one open file, and one of everything. If deadlocks could be eliminated for free, there would not be much discussion. The problem is that the price is high, mostly in terms of putting inconvenient restrictions on processes, as we will see shortly. Thus we are faced with an unpleasant trade-off between convenience and correctness, and a great deal of discussion about which is more important, and to whom. Under these conditions, general solutions are hard to find.

3.4 DEADLOCK DETECTION AND RECOVERY

A second technique is detection and recovery. When this technique is used, the system does not attempt to prevent deadlocks from occurring. Instead, it lets them occur, tries to detect when this happens, and then takes some action to recover after the fact. In this section we will look at some of the ways deadlocks can be detected and some of the ways recovery from them can be handled.

3.4.1 Deadlock Detection with One Resource of Each Type

Let us begin with the simplest case: only one resource of each type exists. Such a system might have one scanner, one CD recorder, one plotter, and one tape drive, but no more than one of each class of resource. In other words, we are excluding systems with two printers for the moment. We will treat them later, using a different method.

For such a system, we can construct a resource graph of the sort illustrated in Fig. 3-3. If this graph contains one or more cycles, a deadlock exists. Any process that is part of a cycle is deadlocked. If no cycles exist, the system is not deadlocked.

As an example of a more complex system than the ones we have looked at so far, consider a system with seven processes, A though G, and six resources, R

through *W*. The state of which resources are currently owned and which ones are currently being requested is as follows:

1. Process *A* holds *R* and wants *S*.
2. Process *B* holds nothing but wants *T*.
3. Process *C* holds nothing but wants *S*.
4. Process *D* holds *U* and wants *S* and *T*.
5. Process *E* holds *T* and wants *V*.
6. Process *F* holds *W* and wants *S*.
7. Process *G* holds *V* and wants *U*.

The question is: "Is this system deadlocked, and if so, which processes are involved?"

To answer this question, we can construct the resource graph of Fig. 3-5(a). This graph contains one cycle, which can be seen by visual inspection. The cycle is shown in Fig. 3-5(b). From this cycle, we can see that processes *D*, *E*, and *G* are all deadlocked. Processes *A*, *C*, and *F* are not deadlocked because *S* can be allocated to any one of them, which then finishes and returns it. Then the other two can take it in turn and also complete.

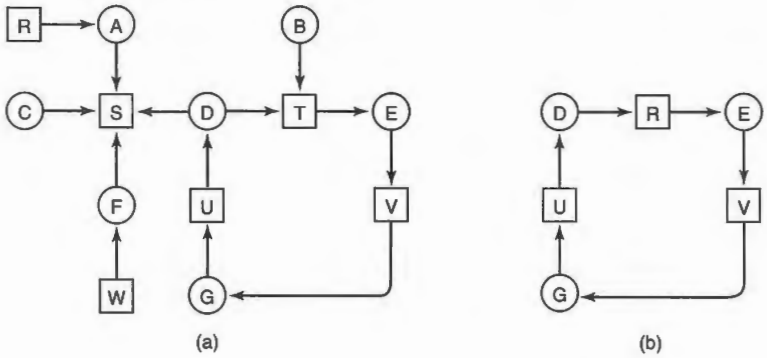


Figure 3-5. (a) A resource graph. (b) A cycle extracted from (a).

Although it is relatively simple to pick out the deadlocked processes by eye from a simple graph, for use in actual systems we need a formal algorithm for detecting deadlocks. Many algorithms for detecting cycles in directed graphs are known. Below we will give a simple one that inspects a graph and terminates either when it has found a cycle or when it has shown that none exist. It uses one data structure, *L*, a list of nodes. During the algorithm, arcs will be marked to indicate that they have already been inspected, to prevent repeated inspections.

The algorithm operates by carrying out the following steps as specified:

1. For each node, N in the graph, perform the following 5 steps with N as the starting node.
2. Initialize L to the empty list, and designate all the arcs as unmarked.
3. Add the current node to the end of L and check to see if the node now appears in L two times. If it does, the graph contains a cycle (listed in L) and the algorithm terminates.
4. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
6. We have now reached a dead end. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 3. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates.

What this algorithm does is take each node, in turn, as the root of what it hopes will be a tree, and does a depth-first search on it. If it ever comes back to a node it has already encountered, then it has found a cycle. If it exhausts all the arcs from any given node, it backtracks to the previous node. If it backtracks to the root and cannot go further, the subgraph reachable from the current node does not contain any cycles. If this property holds for all nodes, the entire graph is cycle free, so the system is not deadlocked.

To see how the algorithm works in practice, let us use it on the graph of Fig. 3-5(a). The order of processing the nodes is arbitrary, so let us just inspect them from left to right, top to bottom, first running the algorithm starting at R, then successively, A, B, C, S, D, T, E, F, and so forth. If we hit a cycle, the algorithm stops.

We start at R and initialize L to the empty list. Then we add R to the list and move to the only possibility, A, and add it to L, giving $L = [R, A]$. From A we go to S, giving $L = [R, A, S]$. S has no outgoing arcs, so it is a dead end, forcing us to backtrack to A. Since A has no unmarked outgoing arcs, we backtrack to R, completing our inspection of R.

Now we restart the algorithm starting at A, resetting L to the empty list. This search, too, quickly stops, so we start again at B. From B we continue to follow outgoing arcs until we get to D, at which time $L = [B, T, E, V, G, U, D]$. Now we must make a (random) choice. If we pick S we come to a dead end and backtrack to D. The second time we pick T and update L to be $[B, T, E, V, G, U, D, T]$, at which point we discover the cycle and stop the algorithm.

This algorithm is far from optimal. For a better one, see (Even, 1979). Nevertheless, it demonstrates that an algorithm for deadlock detection exists.

3.4.2 Deadlock Detection with Multiple Resource of Each Type

When multiple copies of some of the resources exist, a different approach is needed to detect deadlocks. We will now present a matrix-based algorithm for detecting deadlock among n processes, P_1 through P_n. Let the number of resource classes be m, with E_1 resources of class 1, E_2 resources of class 2, and generally, E_i resources of class i ($1 \le i \le m$). E is the **existing resource vector**. It gives the total number of instances of each resource in existence. For example, if class 1 is tape drives, then $E_1 = 2$ means the system has two tape drives.

At any instant, some of the resources are assigned and are not available. Let A be the **available resource vector**, with A_i giving the number of instances of resource i that are currently available (i.e., unassigned). If both of our two tape drives are assigned, A_1 will be 0.

Now we need two arrays, C, the **current allocation matrix**, and R, the **request matrix**. The i-th row of C tells how many instances of each resource class P_i currently holds. Thus C_{ij} is the number of instances of resource j that are held by process i. Similarly, R_{ij} is the number of instances of resource j that P_i wants. These four data structures are shown in Fig. 3-6.

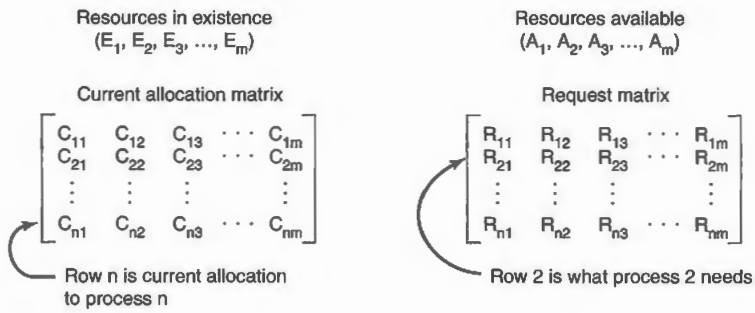


Figure 3-6. The four data structures needed by the deadlock detection algorithm.

An important invariant holds for these four data structures. In particular, every resource is either allocated or is available. This observation means that

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

In other words, if we add up all the instances of the resource j that have been

allocated and to this add all the instances that are available, the result is the number of instances of that resource class that exist.

The deadlock detection algorithm is based on comparing vectors. Let us define the relation $A \leq B$ on two vectors A and B to mean that each element of A is less than or equal to the corresponding element of B. Mathematically, $A \leq B$ holds if and only if $A_i \leq B_i$ for $1 \leq i \leq m$.

Each process is initially said to be unmarked. As the algorithm progresses, processes will be marked, indicating that they are able to complete and are thus not deadlocked. When the algorithm terminates, any unmarked processes are known to be deadlocked.

The deadlock detection algorithm can now be given, as follows.

1. Look for an unmarked process, P_i, for which the i-th row of R is less than or equal to A.
2. If such a process is found, add the i-th row of C to A, mark the process, and go back to step 1.
3. If no such process exists, the algorithm terminates.

When the algorithm finishes, all the unmarked processes, if any, are deadlocked.

What the algorithm is doing in step 1 is looking for a process that can be run to completion. Such a process is characterized as having resource demands that can be met by the currently available resources. The selected process is then run until it finishes, at which time it returns the resources it is holding to the pool of available resources. It is then marked as completed. If all the processes are ultimately able to run, none of them are deadlocked. If some of them can never run, they are deadlocked. Although the algorithm is nondeterministic (because it may run the processes in any feasible order), the result is always the same.

As an example of how the deadlock detection algorithm works, consider Fig. 3-7. Here we have three processes and four resource classes, which we have arbitrarily labeled tape drives, plotters, scanner, and CD-ROM drive. Process 1 has one scanner. Process 2 has two tape drives and a CD-ROM drive. Process 3 has a plotter and two scanners. Each process needs additional resources, as shown by the R matrix .

To run the deadlock detection algorithm, we look for a process whose resource request can be satisfied. The first one cannot be satisfied because there is no CD-ROM drive available. The second cannot be satisfied either, because there is no scanner free. Fortunately, the third one can be satisfied, so process 3 runs and eventually returns all its resources, giving

$$A = (2\ 2\ 2\ 0)$$

At this point process 2 can run and return its resources, giving

$$A = (4\ 2\ 2\ 1)$$

Now the remaining process can run. There is no deadlock in the system.

$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (4 & 2 & 3 & 1) \end{array} \qquad A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (2 & 1 & 0 & 0) \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Figure 3-7. An example for the deadlock detection algorithm.

Now consider a minor variation of the situation of Fig. 3-7. Suppose that process 2 needs a CD-ROM drive as well as the two tape drives and the plotter. None of the requests can be satisfied, so the entire system is deadlocked.

Now that we know how to detect deadlocks, the question of when to look for them comes up. One possibility is to check every time a resource request is made. This is certain to detect them as early as possible, but it is potentially expensive in terms of CPU time. An alternative strategy is to check every k minutes, or perhaps only when the CPU utilization has dropped below some threshold. The reason for considering the CPU utilization is that if enough processes are deadlocked, there will be few runnable processes, and the CPU will often be idle.

3.4.3 Recovery from Deadlock

Suppose that our deadlock detection algorithm has succeeded and detected a deadlock. What next? Some way is needed to recover and get the system going again. In this section we will discuss various ways of recovering from deadlock. None of them are especially attractive, however.

Recovery through Preemption

In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process. In many cases, manual intervention may be required, especially in batch processing operating systems running on mainframes.

For example, to take a laser printer away from its owner, the operator can collect all the sheets already printed and put them in a pile. Then the process can be suspended (marked as not runnable). At this point the printer can be assigned to

another process. When that process finishes, the pile of printed sheets can be put back in the printer's output tray and the original process restarted.

The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource. Recovering this way is frequently difficult or impossible. Choosing the process to suspend depends largely on which ones have resources that can easily be taken back.

Recovery through Rollback

If the system designers and machine operators know that deadlocks are likely, they can arrange to have processes **checkpointed** periodically. Checkpointing a process means that its state is written to a file so that it can be restarted later. The checkpoint contains not only the memory image, but also the resource state, that is, which resources are currently assigned to the process. To be most effective, new checkpoints should not overwrite old ones but should be written to new files, so as the process executes, a whole sequence of checkpoint files are accumulated.

When a deadlock is detected, it is easy to see which resources are needed. To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired some other resource by starting one of its earlier checkpoints. All the work done since the checkpoint is lost (e.g., output printed since the checkpoint must be discarded, since it will be printed again). In effect, the process is reset to an earlier moment when it did not have the resource, which is now assigned to one of the deadlocked processes. If the restarted process tries to acquire the resource again, it will have to wait until it becomes available.

Recovery through Killing Processes

The crudest, but simplest way to break a deadlock is to kill one or more processes. One possibility is to kill a process in the cycle. With a little luck, the other processes will be able to continue. If this does not help, it can be repeated until the cycle is broken.

Alternatively, a process not in the cycle can be chosen as the victim in order to release its resources. In this approach, the process to be killed is carefully chosen because it is holding resources that some process in the cycle needs. For example, one process might hold a printer and want a plotter, with another process holding a plotter and wanting a printer. These two are deadlocked. A third process may hold another identical printer and another identical plotter and be happily running. Killing the third process will release these resources and break the deadlock involving the first two.

Where possible, it is best to kill a process that can be rerun from the beginning with no ill effects. For example, a compilation can always be rerun because

all it does is read a source file and produce an object file. If it is killed part way through, the first run has no influence on the second run.

On the other hand, a process that updates a database cannot always be run a second time safely. If the process adds 1 to some record in the database, running it once, killing it, and then running it again will add 2 to the record, which is incorrect.

3.5 DEADLOCK AVOIDANCE

In the discussion of deadlock detection, we tacitly assumed that when a process asks for resources, it asks for them all at once (the *R* matrix of Fig. 3-6). In most systems, however, resources are requested one at a time. The system must be able to decide whether granting a resource is safe or not and only make the allocation when it is safe. Thus the question arises: Is there an algorithm that can always avoid deadlock by making the right choice all the time? The answer is a qualified yes—we can avoid deadlocks, but only if certain information is available in advance. In this section we examine ways to avoid deadlock by careful resource allocation.

3.5.1 Resource Trajectories

The main algorithms for doing deadlock avoidance are based on the concept of safe states. Before describing the algorithms, we will make a slight digression to look at the concept of safety in a graphic and easy-to-understand way. Although the graphical approach does not translate directly into a usable algorithm, it gives a good intuitive feel for the nature of the problem.

In Fig. 3-8 we see a model for dealing with two processes and two resources, for example, a printer and a plotter. The horizontal axis represents the number of instructions executed by process *A*. The vertical axis represents the number of instructions executed by process *B*. At I_1 *A* requests a printer; at I_2 it needs a plotter. The printer and plotter are released at I_3 and I_4, respectively. Process *B* needs the plotter from I_5 to I_7 and the printer from I_6 to I_8.

Every point in the diagram represents a joint state of the two processes. Initially, the state is at *p*, with neither process having executed any instructions. If the scheduler chooses to run *A* first, we get to the point *q*, in which *A* has executed some number of instructions, but *B* has executed none. At point *q* the trajectory becomes vertical, indicating that the scheduler has chosen to run *B*. With a single processor, all paths must be horizontal or vertical, never diagonal. Furthermore, motion is always to the north or east, never to the south or west (processes cannot run backward).

When *A* crosses the I_1 line on the path from *r* to *s*, it requests and is granted the printer. When *B* reaches point *t*, it requests the plotter.

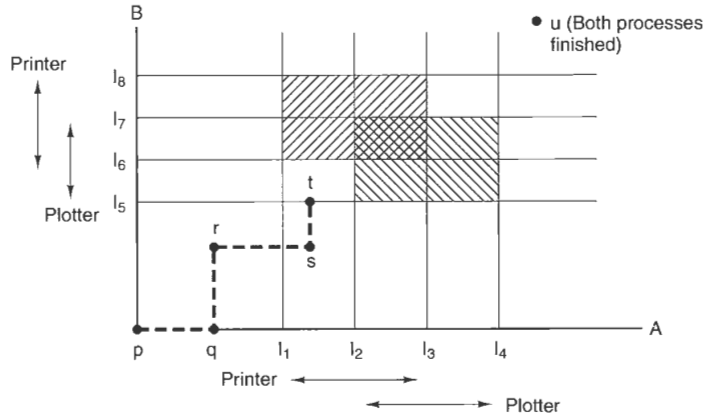


Figure 3-8. Two process resource trajectories.

The regions that are shaded are especially interesting. The region with lines slanting from southwest to northeast represents both processes having the printer. The mutual exclusion rule makes it impossible to enter this region. Similarly, the region shaded the other way represents both processes having the plotter, and is equally impossible.

If the system ever enters the box bounded by I_1 and I_2 on the sides and I_5 and I_6 top and bottom, it will eventually deadlock when it gets to the intersection of I_2 and I_6. At this point, *A* is requesting the plotter and *B* is requesting the printer, and both are already assigned. The entire box is unsafe and must not be entered. At point *t* the only safe thing to do is run process *A* until it gets to I_4. Beyond that, any trajectory to *u* will do.

The important thing to see here is at point *t B* is requesting a resource. The system must decide whether to grant it or not. If the grant is made, the system will enter an unsafe region and eventually deadlock. To avoid the deadlock, *B* should be suspended until *A* has requested and released the plotter.

3.5.2 Safe and Unsafe States

The deadlock avoidance algorithms that we will study use the information of Fig. 3-6. At any instant of time, there is a current state consisting of *E*, *A*, *C*, and *R*. A state is said to be **safe** if it is not deadlocked and there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately. It is easiest to illustrate this concept by an example using one resource. In Fig. 3-9(a) we have a state in which *A* has 3 instances of the resource but may need as many as 9 eventually. *B* currently has 2 and may need 4 altogether, later. Similarly, *C* also has 2 but may

need an additional 5. A total of 10 instances of the resource exist, so with 7 resources already allocated, there are 3 still free.

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0
(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7
(e)

Figure 3-9. Demonstration that the state in (a) is safe.

The state of Fig. 3-9(a) is safe because there exists a sequence of allocations that allows all processes to complete. Namely, the scheduler could simply run *B* exclusively, until it asked for and got two more instances of the resource, leading to the state of Fig. 3-9(b). When *B* completes, we get the state of Fig. 3-9(c). Then the scheduler can run *C*, leading eventually to Fig. 3-9(d). When *C* completes, we get Fig. 3-9(e). Now *A* can get the six instances of the resource it needs and also complete. Thus the state of Fig. 3-9(a) is safe because the system, by careful scheduling, can avoid deadlock.

Now suppose we have the initial state shown in Fig. 3-10(a), but this time *A* requests and gets another resource, giving Fig. 3-10(b). Can we find a sequence that is guaranteed to work? Let us try. The scheduler could run *B* until it asked for all its resources, as shown in Fig. 3-10(c).

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2
(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0
(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4
(d)

Figure 3-10. Demonstration that the state in (b) is not safe.

Eventually, *B* completes and we get the situation of Fig. 3-10(d). At this point we are stuck. We only have four instances of the resource free, and each of the active processes needs five. There is no sequence that guarantees completion. Thus the allocation decision that moved the system from Fig. 3-10(a) to Fig. 3-10(b) went from a safe state to an unsafe state. Running *A* or *C* next starting at Fig. 3-10(b) does not work either. In retrospect, *A*'s request should not have been granted.

It is worth noting that an unsafe state is not a deadlocked state. Starting at Fig. 3-10(b), the system can run for a while. In fact, one process can even complete. Furthermore, it is possible that *A* might release a resource before asking for

any more, allowing *C* to complete and avoiding deadlock altogether. Thus the difference between a safe state and an unsafe state is that from a safe state the system can *guarantee* that all processes will finish; from an unsafe state, no such guarantee can be given.

3.5.3 The Banker's Algorithm for a Single Resource

A scheduling algorithm that can avoid deadlocks is due to Dijkstra (1965) and is known as the **banker's algorithm** and is an extension of the deadlock detection algorithm given in Sec. 3.4.1. It is modeled on the way a small-town banker might deal with a group of customers to whom he has granted lines of credit. What the algorithm does is check to see if granting the request leads to an unsafe state. If it does, the request is denied. If granting the request leads to a safe state, it is carried out. In Fig. 3-11(a) we see four customers, *A*, *B*, *C*, and *D*, each of whom has been granted a certain number of credit units (e.g., 1 unit is 1K dollars). The banker knows that not all customers will need their maximum credit immediately, so he has reserved only 10 units rather than 22 to service them. (In this analogy, customers are processes, units are, say, tape drives, and the banker is the operating system.)

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

Figure 3-11. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

The customers go about their respective businesses, making loan requests from time to time (i.e., asking for resources). At a certain moment, the situation is as shown in Fig. 3-11(b). This state is safe because with two units left, the banker can delay any requests except *C*'s, thus letting *C* finish and release all four of his resources. With four units in hand, the banker can let either *D* or *B* have the necessary units, and so on.

Consider what would happen if a request from *B* for one more unit were granted in Fig. 3-11(b). We would have situation Fig. 3-11(c), which is unsafe. If all the customers suddenly asked for their maximum loans, the banker could not satisfy any of them, and we would have a deadlock. An unsafe state does not *have* to lead to deadlock, since a customer might not need the entire credit line available, but the banker cannot count on this behavior.

The banker's algorithm considers each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted; otherwise, it is postponed

until later. To see if a state is safe, the banker checks to see if he has enough resources to satisfy some customer. If so, those loans are assumed to be repaid, and the customer now closest to the limit is checked, and so on. If all loans can eventually be repaid, the state is safe and the initial request can be granted.

3.5.4 The Banker's Algorithm for Multiple Resources

The banker's algorithm can be generalized to handle multiple resources. Figure 3-12 shows how it works.

Process	Tape drives	Plotters	Scanners	CD ROMs
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Scanners	CD ROMs
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

Figure 3-12. The banker's algorithm with multiple resources.

In Fig. 3-12 we see two matrices. The one on the left shows how many of each resource are currently assigned to each of the five processes. The matrix on the right shows how many resources each process still needs in order to complete. These matrices are just *C* and *R* from Fig. 3-6. As in the single resource case, processes must state their total resource needs before executing, so that the system can compute the right-hand matrix at each instant.

The three vectors at the right of the figure show the existing resources, *E*, the possessed resources, *P*, and the available resources, *A*, respectively. From *E* we see that the system has six tape drives, three plotters, four printers, and two CD-ROM drives. Of these, five tape drives, three plotters, two printers, and two CD-ROM drives are currently assigned. This fact can be seen by adding up the four resource columns in the left-hand matrix. The available resource vector is simply the difference between what the system has and what is currently in use.

The algorithm for checking to see if a state is safe can now be stated.

1. Look for a row, *R*, whose unmet resource needs are all smaller than or equal to *A*. If no such row exists, the system will eventually deadlock since no process can run to completion.

2. Assume the process of the row chosen requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to the *A* vector.

3. Repeat steps 1 and 2 until either all processes are marked terminated, in which case the initial state was safe, or until a deadlock occurs, in which case it was not.

If several processes are eligible to be chosen in step 1, it does not matter which one is selected: the pool of available resources either gets larger, or at worst, stays the same.

Now let us get back to the example of Fig. 3-12. The current state is safe. Suppose that process *B* now requests a printer. This request can be granted because the resulting state is still safe (process *D* can finish, and then processes *A* or *E*, followed by the rest).

Now imagine that after giving *B* one of the two remaining printers, *E* wants the last printer. Granting that request would reduce the vector of available resources to (1 0 0 0), which leads to deadlock. Clearly *E*'s request must be deferred for a while.

The banker's algorithm was first published by Dijkstra in 1965. Since that time, nearly every book on operating systems has described it in detail. Innumerable papers have been written about various aspects of it. Unfortunately, few authors have had the audacity to point out that although in theory the algorithm is wonderful, in practice it is essentially useless because processes rarely know in advance what their maximum resource needs will be. In addition, the number of processes is not fixed, but dynamically varying as new users log in and out. Furthermore, resources that were thought to be available can suddenly vanish (tape drives can break). Thus in practice, few, if any, existing systems use the banker's algorithm for avoiding deadlocks.

3.6 DEADLOCK PREVENTION

Having seen that deadlock avoidance is essentially impossible, because it requires information about future requests, which is not known, how do real systems avoid deadlock? The answer is to go back to the four conditions stated by Coffman et al. (1971) to see if they can provide a clue. If we can ensure that at least one of these conditions is never satisfied, then deadlocks will be structurally impossible (Havender, 1968).

3.6.1 Attacking the Mutual Exclusion Condition

First let us attack the mutual exclusion condition. If no resource were ever assigned exclusively to a single process, we would never have deadlocks. However, it is equally clear that allowing two processes to write on the printer at the

same time will lead to chaos. By spooling printer output, several processes can generate output at the same time. In this model, the only process that actually requests the physical printer is the printer daemon. Since the daemon never requests any other resources, we can eliminate deadlock for the printer.

Unfortunately, not all devices can be spooled (the process table does not lend itself well to being spooled). Furthermore, competition for disk space for spooling can itself lead to deadlock. What would happen if two processes each filled up half of the available spooling space with output and neither was finished producing output? If the daemon was programmed to begin printing even before all the output was spooled, the printer might lie idle if an output process decided to wait several hours after the first burst of output. For this reason, daemons are normally programmed to print only after the complete output file is available. In this case we have two processes that have each finished part, but not all, of their output, and cannot continue. Neither process will ever finish, so we have a deadlock on the disk.

Nevertheless, there is a germ of an idea here that is frequently applicable. Avoid assigning a resource when that is not absolutely necessary, and try to make sure that as few processes as possible may actually claim the resource.

3.6.2 Attacking the Hold and Wait Condition

The second of the conditions stated by Coffman et al. looks slightly more promising. If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks. One way to achieve this goal is to require all processes to request all their resources before starting execution. If everything is available, the process will be allocated whatever it needs and can run to completion. If one or more resources are busy, nothing will be allocated and the process would just wait.

An immediate problem with this approach is that many processes do not know how many resources they will need until they have started running. In fact, if they knew, the banker's algorithm could be used. Another problem is that resources will not be used optimally with this approach. Take, as an example, a process that reads data from an input tape, analyzes it for an hour, and then writes an output tape as well as plotting the results. If all resources must be requested in advance, the process will tie up the output tape drive and the plotter for an hour.

Nevertheless, some mainframe batch systems require the user to list all the resources on the first line of each job. The system then acquires all resources immediately and keeps them until the job finishes. While this method puts a burden on the programmer and wastes resources, it does prevent deadlocks.

A slightly different way to break the hold-and-wait condition is to require a process requesting a resource to first temporarily release all the resources it currently holds. Then it tries to get everything it needs all at once.

3.6.3 Attacking the No Preemption Condition

Attacking the third condition (no preemption) is even less promising than attacking the second one. If a process has been assigned the printer and is in the middle of printing its output, forcibly taking away the printer because a needed plotter is not available is tricky at best and impossible at worst.

3.6.4 Attacking the Circular Wait Condition

Only one condition is left. The circular wait can be eliminated in several ways. One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one. For a process that needs to copy a huge file from a tape to a printer, this restriction is unacceptable.

Another way to avoid the circular wait is to provide a global numbering of all the resources, as shown in Fig. 3-13(a). Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first a printer and then a tape drive, but it may not request first a plotter and then a printer.

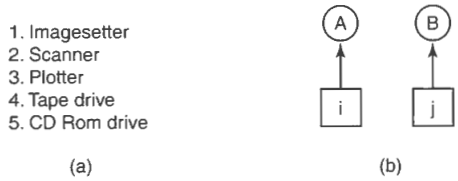


Figure 3-13. (a) Numerically ordered resources. (b) A resource graph.

With this rule, the resource allocation graph can never have cycles. Let us see why this is true for the case of two processes, in Fig. 3-13(b). We can get a deadlock only if A requests resource j and B requests resource i. Assuming i and j are distinct resources, they will have different numbers. If $i > j$, then A is not allowed to request j because that is lower than what it already has. If $i < j$, then B is not allowed to request i because that is lower than what it already has. Either way, deadlock is impossible.

With multiple processes, the same logic holds. At every instant, one of the assigned resources will be highest. The process holding that resource will never ask for a resource already assigned. It will either finish, or at worst, request even higher numbered resources, all of which are available. Eventually, it will finish and free its resources. At this point, some other process will hold the highest resource and can also finish. In short, there exists a scenario in which all processes finish, so no deadlock is present.

A minor variation of this algorithm is to drop the requirement that resources be acquired in strictly increasing sequence and merely insist that no process request a resource lower than what it is already holding. If a process initially requests 9 and 10, and then releases both of them, it is effectively starting all over, so there is no reason to prohibit it from now requesting resource 1.

Although numerically ordering the resources eliminates the problem of deadlocks, it may be impossible to find an ordering that satisfies everyone. When the resources include process table slots, disk spooler space, locked database records, and other abstract resources, the number of potential resources and different uses may be so large that no ordering could possibly work.

The various approaches to deadlock prevention are summarized in Fig. 3-14.

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Figure 3-14. Summary of approaches to deadlock prevention.

3.7 OTHER ISSUES

In this section we will discuss a few miscellaneous issues related to deadlocks. These include two-phase locking, nonresource deadlocks, and starvation.

3.7.1 Two-Phase Locking

Although both avoidance and prevention are not terribly promising in the general case, for specific applications, many excellent special-purpose algorithms are known. As an example, in many database systems, an operation that occurs frequently is requesting locks on several records and then updating all the locked records. When multiple processes are running at the same time, there is a real danger of deadlock.

The approach often used is called **two-phase locking**. In the first phase, the process tries to lock all the records it needs, one at a time. If it succeeds, it begins the second phase, performing its updates and releasing the locks. No real work is done in the first phase.

If during the first phase, some record is needed that is already locked, the process just releases all its locks and starts the first phase all over. In a certain

sense, this approach is similar to requesting all the resources needed in advance, or at least before anything irreversible is done. In some versions of two-phase locking, there is no release and restart if a lock is encountered during the first phase. In these versions, deadlock can occur.

However, this strategy is not applicable in general. In real-time systems and process control systems, for example, it is not acceptable to just terminate a process partway through because a resource is not available and start all over again. Neither is it acceptable to start over if the process has read or written messages to the network, updated files, or anything else that cannot be safely repeated. The algorithm works only in those situations where the programmer has very carefully arranged things so that the program can be stopped at any point during the first phase and restarted. Many applications cannot be structured this way.

3.7.2 Nonresource Deadlocks

All of our work so far has concentrated on resource deadlocks. One process wants something that another process has and must wait until the first one gives it up. Deadlocks can also occur in other situations, however, including those not involving resources at all.

For example, it can happen that two processes deadlock each waiting for the other one to do something. This often happens with semaphores. In Chap. 2 we saw examples in which a process had to do a down on two semaphores, typically *mutex* and another one. If these are done in the wrong order, deadlock can result.

3.7.3 Starvation

A problem closely related to deadlock is **starvation**. In a dynamic system, requests for resources happen all the time. Some policy is needed to make a decision about who gets which resource when. This policy, although seemingly reasonable, may lead to some processes never getting service even though they are not deadlocked.

As an example, consider allocation of the printer. Imagine that the system uses some kind of algorithm to ensure that allocating the printer does not lead to deadlock. Now suppose that several processes all want it at once. Which one should get it?

One possible allocation algorithm is to give it to the process with the smallest file to print (assuming this information is available). This approach maximizes the number of happy customers and seems fair. Now consider what happens in a busy system when one process has a huge file to print. Every time the printer is free, the system will look around and choose the process with the shortest file. If there is a constant stream of processes with short files, the process with the huge

file will never be allocated the printer. It will simply starve to death (be postponed indefinitely, even though it is not blocked).

Starvation can be avoided by using a first-come, first-serve, resource allocation policy. With this approach, the process waiting the longest gets served next. In due course of time, any given process will eventually become the oldest and thus get the needed resource.

3.8 RESEARCH ON DEADLOCKS

If ever there was a subject that was investigated mercilessly during the early days of operating systems, it was deadlocks. The reason for this is that deadlock detection is a nice little graph theory problem that one mathematically-inclined graduate student can get his jaws around and chew on for 3 or 4 years. All kinds of algorithms were devised, each one more exotic and less practical than the previous one. Essentially, all this research has died out, with only a very occasional new paper appearing (e.g., Karacali et al., 2000). When an operating system wants to do deadlock detection or prevention, which few of them do, they use one of the methods discussed in this chapter,

There is still a little research on distributed deadlock detection, however. We will not treat that here because (1) it is outside the scope of this book, and (2) none of it is even remotely practical in real systems. Its main function seems to be keeping otherwise unemployed graph theorists off the streets.

3.9 SUMMARY

Deadlock is a potential problem in any operating system. It occurs when a group of processes each have been granted exclusive access to some resources, and each one wants yet another resource that belongs to another process in the group. All of them are blocked and none will ever run again.

Deadlock can be avoided by keeping track of which states are safe and which are unsafe. A safe state is one in which there exists a sequence of events that guarantee that all processes can finish. An unsafe state has no such guarantee. The banker's algorithm avoids deadlock by not granting a request if that request will put the system in an unsafe state.

Deadlock can be structurally prevented by building the system in such a way that it can never occur by design. For example, by allowing a process to hold only one resource at any instant the circular wait condition required for deadlock is broken. Deadlock can also be prevented by numbering all the resources, and making processes request them in strictly increasing order. Starvation can be avoided by a first-come, first-served allocation policy.

PROBLEMS

1. Give an example of a deadlock taken from politics.
2. Students working at individual PCs in a computer laboratory send their files to be printed by a server which spools the files on its hard disk. Under what conditions may a deadlock occur if the disk space for the print spool is limited? How may the deadlock be avoided?
3. In the preceding question which resources are preemptable and which are nonpreemptable?
4. In Fig. 3-1 the resources are returned in the reverse order of their acquisition. Would giving them back in the other order be just as good?
5. Fig. 3-3 shows the concept of a resource graph. Do illegal graphs exist, that is graphs that structurally violate the model we have used of resource usage? If so, give an example of one.
6. The discussion of the ostrich algorithm mentions the possibility of process table slots or other system tables filling up. Can you suggest a way to enable a system administrator to recover from such a situation?
7. Consider Fig. 3-4. Suppose that in step (o) C requested S instead of requesting R. Would this lead to deadlock? Suppose that it requested both S and R?
8. At a crossroads with STOP signs on all four approaches, the rule is that each driver yields the right of way to the driver on his right. This rule is not adequate when four vehicles arrive simultaneously. Fortunately, humans are sometimes capable of acting more intelligently than computers and the problem is usually resolved when one driver signals the driver to his left to go ahead. Can you draw an analogy between this behavior and any of the ways of recovering from deadlock described in Sec. 3.4.3? Why is a problem with such a simple solution in the human world so difficult to apply to a computer system?
9. Suppose that in Fig. 3-6 $C_{ij} + R_{ij} > E_j$ for some i. What implications does this have for all the processes finishing without deadlock?
10. All the trajectories in Fig. 3-8 are horizontal or vertical. Can you envision any circumstances in which diagonal trajectories were also possible?
11. Can the resource trajectory scheme of Fig. 3-8 also be used to illustrate the problem of deadlocks with three processes and three resources? If so, how can this be done? If not, why not?
12. In theory, resource trajectory graphs could be used to avoid deadlocks. By clever scheduling, the operating system could avoid unsafe regions. Suggest a practical problem with actually doing this.
13. Take a careful look at Fig. 3-11(b). If D asks for one more unit, does this lead to a safe state or an unsafe one? What if the request came from C instead of D?
14. Can a system be in a state that is neither deadlocked nor safe? If so, give an example. If not, prove that all states are either deadlocked or safe.

15. A system has two processes and three identical resources. Each process needs a maximum of two resources. Is deadlock possible? Explain your answer.

16. Consider the previous problem again, but now with p processes each needing a maximum of m resources and a total of r resources available. What condition must hold to make the system deadlock free?

17. Suppose that process A in Fig. 3-12 requests the last tape drive. Does this action lead to a deadlock?

18. A computer has six tape drives, with n processes competing for them. Each process may need two drives. For which values of n is the system deadlock free?

19. The banker's algorithm is being run in a system with m resource classes and n processes. In the limit of large m and n, the number of operations that must be performed to check a state for safety is proportional to $m^a n^b$. What are the values of a and b?

20. A system has four processes and five allocatable resources. The current allocation and maximum needs are as follows:

	Allocated	*Maximum*	*Available*
Process A	1 0 2 1 1	1 1 2 1 3	0 0 x 1 1
Process B	2 0 1 1 0	2 2 2 1 0	
Process C	1 1 0 1 0	2 1 3 1 0	
Process D	1 1 1 1 0	1 1 2 2 1	

What is the smallest value of x for which this is a safe state?

21. A distributed system using mailboxes has two IPC primitives, send and receive. The latter primitive specifies a process to receive from and blocks if no message from that process is available, even though messages may be waiting from other processes. There are no shared resources, but processes need to communicate frequently about other matters. Is deadlock possible? Discuss.

22. Two processes, A and B, each need three records, 1, 2, and 3, in a database. If A asks for them in the order 1, 2, 3, and B asks for them in the same order, deadlock is not possible. However, if B asks for them in the order 3, 2, 1, then deadlock is possible. With three resources, there are 3! or 6 possible combinations each process can request the resources. What fraction of all the combinations are guaranteed to be deadlock free?

23. Now reconsider the above problem, but using two-phase locking. Will that eliminate the potential for deadlock? Does it have any other undesirable characteristics, however? If so, which ones?

24. In an electronic funds transfer system, there are hundreds of identical processes that work as follows. Each process reads an input line specifying an amount of money, the account to be credited, and the account to be debited. Then it locks both accounts and transfers the money, releasing the locks when done. With many processes running in parallel, there is a very real danger that having locked account x it will be unable to lock y because y has been locked by a process now waiting for x. Devise a scheme that avoids deadlocks. Do not release an account record until you have completed the

transactions. (In other words, solutions that lock one account and then release it immediately if the other is locked are not allowed.)

25. One way to prevent deadlocks is to eliminate the hold-and-wait condition. In the text it was proposed that before asking for a new resource, a process must first release whatever resources it already holds (assuming that is possible). However, doing so introduces the danger that it may get the new resource but lose some of the existing ones to competing processes. Propose an improvement to this scheme.

26. A computer science student assigned to work on deadlocks thinks of the following brilliant way to eliminate deadlocks. When a process requests a resource, it specifies a time limit. If the process blocks because the resource is not available, a timer is started. If the time limit is exceeded, the process is released and allowed to run again. If you were the professor, what grade would you give this proposal and why.

27. Cinderella and the Prince are getting divorced. To divide their property, they have agreed on the following algorithm. Every morning, each one may send a letter to the other's lawyer requesting one item of property. Since it takes a day for letters to be delivered, they have agreed that if both discover that they have requested the same item on the same day, the next day they will send a letter canceling the request. Among their property is their dog, Woofer, Woofer's doghouse, their canary, Tweeter, and Tweeter's cage. The animals love their houses, so it has been agreed that any division of property separating an animal from its house is invalid, requiring the whole division to start over from scratch. Both Cinderella and the Prince desperately want Woofer. So they can go on (separate) vacations, each spouse has programmed a personal computer to handle the negotiation. When they come back from vacation, the computers are still negotiating. Why? Is deadlock possible? Is starvation possible? Discuss.

28. A student majoring in anthropology and minoring in computer science has embarked on a research project to see if African baboons can be taught about deadlocks. He locates a deep canyon and fastens a rope across it, so the baboons can cross hand-over-hand. Several baboons can cross at the same time, provided that they are all going in the same direction. If eastward moving and westward moving baboons ever get onto the rope at the same time, a deadlock will result (the baboons will get stuck in the middle) because it is impossible for one baboon to climb over another one while suspended over the canyon. If a baboon wants to cross the canyon, he must check to see that no other baboon is currently crossing in the opposite direction. Write a program using semaphores that avoids deadlock. Do not worry about a series of eastward moving baboons holding up the westward moving baboons indefinitely.

29. Repeat the previous problem, but now avoid starvation. When a baboon that wants to cross to the east arrives at the rope and finds baboons crossing to the west, he waits until the rope is empty, but no more westward moving baboons are allowed to start until at least one baboon has crossed the other way.

30. Program a simulation of the banker's algorithm. Your program should cycle through each of the bank clients asking for a request and evaluating whether it is safe or unsafe. Output a log of requests and decisions to a file.

4

MEMORY MANAGEMENT

Memory is an important resource that must be carefully managed. While the average home computer nowadays has a thousand times as much memory as the IBM 7094, the largest computer in the world in the early 1960s, programs are getting bigger faster than memories. To paraphrase Parkinson's law, "Programs expand to fill the memory available to hold them." In this chapter we will study how operating systems manage memory.

Ideally, what every programmer would like is an infinitely large, infinitely fast memory that is also nonvolatile, that is, does not lose its contents when the electric power fails. While we are at it, why not also ask for it to be inexpensive, too? Unfortunately technology does not provide such memories. Consequently, most computers have a **memory hierarchy**, with a small amount of very fast, expensive, volatile cache memory, tens of megabytes of medium-speed, medium-price, volatile main memory (RAM), and tens or hundreds of gigabytes of slow, cheap, nonvolatile disk storage. It is the job of the operating system to coordinate how these memories are used.

The part of the operating system that manages the memory hierarchy is called the **memory manager**. Its job is to keep track of which parts of memory are in use and which parts are not in use, to allocate memory to processes when they need it and deallocate it when they are done, and to manage swapping between main memory and disk when main memory is too small to hold all the processes.

In this chapter we will investigate a number of different memory management schemes, ranging from very simple to highly sophisticated. We will start at the

IBM Ex-1011
Page 213

beginning and look first at the simplest possible memory management system and then gradually progress to more and more elaborate ones.

As we pointed out in Chap. 1, history tends to repeat itself in the computer world. While the simplest memory management schemes are no longer used on desktop computers, they are still used in some palmtop, embedded, and smart card systems. For this reason, they are still worth studying.

4.1 BASIC MEMORY MANAGEMENT

Memory management systems can be divided into two classes: those that move processes back and forth between main memory and disk during execution (swapping and paging), and those that do not. The latter are simpler, so we will study them first. Later in the chapter we will examine swapping and paging. Throughout this chapter the reader should keep in mind that swapping and paging are largely artifacts caused by the lack of sufficient main memory to hold all the programs at once. If main memory ever gets so large that there is truly enough of it, the arguments in favor of one kind of memory management scheme or another may become obsolete.

On the other hand, as mentioned above, software seems to be growing even faster than memory, so efficient memory management may always be needed. In the 1980s, there were many universities that ran a timesharing system with dozens of (more-or-less satisfied) users on a 4 MB VAX. Now Microsoft recommends having at least 64 MB for a single-user Windows 2000 system. The trend toward multimedia puts even more demands on memory, so good memory management is probably going to be needed for the next decade at least.

4.1.1 Monoprogramming without Swapping or Paging

The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the operating system. Three variations on this theme are shown in Fig. 4-1. The operating system may be at the bottom of memory in RAM (Random Access Memory), as shown in Fig. 4-1(a), or it may be in ROM (Read-Only Memory) at the top of memory, as shown in Fig. 4-1(b), or the device drivers may be at the top of memory in a ROM and the rest of the system in RAM down below, as shown in Fig. 4-1(c). The first model was formerly used on mainframes and minicomputers but is rarely used any more. The second model is used on some palmtop computers and embedded systems. The third model was used by early personal computers (e.g., running MS-DOS), where the portion of the system in the ROM is called the **BIOS** (Basic Input Output System).

When the system is organized in this way, only one process at a time can be running. As soon as the user types a command, the operating system copies the

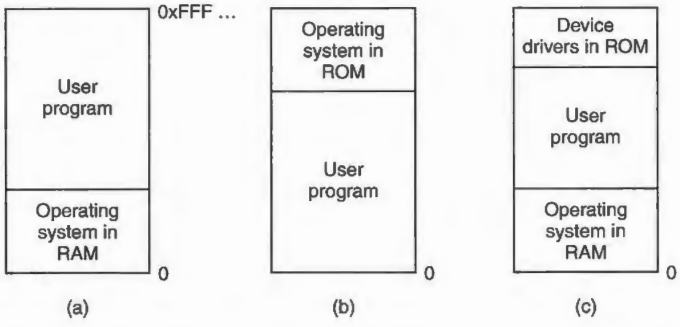


Figure 4-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

requested program from disk to memory and executes it. When the process finishes, the operating system displays a prompt character and waits for a new command. When it receives the command, it loads a new program into memory, overwriting the first one.

4.1.2 Multiprogramming with Fixed Partitions

Except on simple embedded systems, monoprogramming is hardly used any more. Most modern systems allow multiple processes to run at the same time. Having multiple processes running at once means that when one process is blocked waiting for I/O to finish, another one can use the CPU. Thus multiprogramming increases the CPU utilization. Network servers always have the ability to run multiple processes (for different clients) at the same time, but most client (i.e., desktop) machines also have this ability nowadays.

The easiest way to achieve multiprogramming is simply to divide memory up into n (possibly unequal) partitions. This partitioning can, for example, be done manually when the system is started up.

When a job arrives, it can be put into the input queue for the smallest partition large enough to hold it. Since the partitions are fixed in this scheme, any space in a partition not used by a job is lost. In Fig. 4-2(a) we see how this system of fixed partitions and separate input queues looks.

The disadvantage of sorting the incoming jobs into separate queues becomes apparent when the queue for a large partition is empty but the queue for a small partition is full, as is the case for partitions 1 and 3 in Fig. 4-2(a). Here small jobs have to wait to get into memory, even though plenty of memory is free. An alternative organization is to maintain a single queue as in Fig. 4-2(b). Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run. Since it is undesirable to waste a large

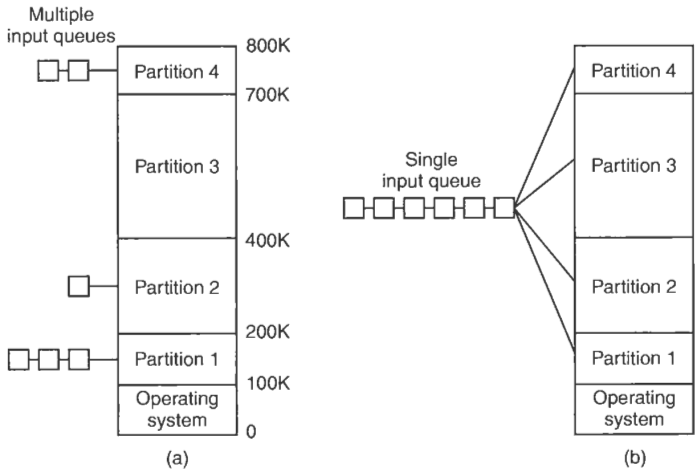


Figure 4-2. (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

partition on a small job, a different strategy is to search the whole input queue whenever a partition becomes free and pick the largest job that fits. Note that the latter algorithm discriminates against small jobs as being unworthy of having a whole partition, whereas usually it is desirable to give the smallest jobs (often interactive jobs) the best service, not the worst.

One way out is to have at least one small partition around. Such a partition will allow small jobs to run without having to allocate a large partition for them.

Another approach is to have a rule stating that a job that is eligible to run may not be skipped over more than k times. Each time it is skipped over, it gets one point. When it has acquired k points, it may not be skipped again.

This system, with fixed partitions set up by the operator in the morning and not changed thereafter, was used by OS/360 on large IBM mainframes for many years. It was called **MFT** (Multiprogramming with a Fixed number of Tasks or OS/MFT). It is simple to understand and equally simple to implement: incoming jobs are queued until a suitable partition is available, at which time the job is loaded into that partition and run until it terminates. Nowadays, few, if any, operating systems, support this model.

4.1.3 Modeling Multiprogramming

When multiprogramming is used, the CPU utilization can be improved. Crudely put, if the average process computes only 20 percent of the time it is sitting in memory, with five processes in memory at once, the CPU should be busy

all the time. This model is unrealistically optimistic, however, since it assumes that all five processes will never be waiting for I/O at the same time.

A better model is to look at CPU usage from a probabilistic viewpoint. Suppose that a process spends a fraction p of its time waiting for I/O to complete. With n processes in memory at once, the probability that all n processes are waiting for I/O (in which case the CPU will be idle) is p^n. The CPU utilization is then given by the formula

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

Figure 4-3 shows the CPU utilization as a function of n, which is called the **degree of multiprogramming**.

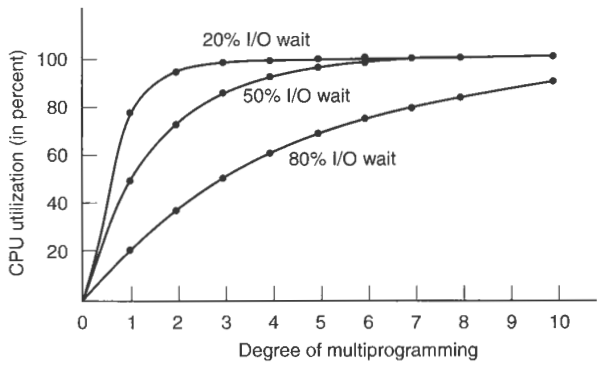


Figure 4-3. CPU utilization as a function of the number of processes in memory.

From the figure it is clear that if processes spend 80 percent of their time waiting for I/O, at least 10 processes must be in memory at once to get the CPU waste below 10 percent. When you realize that an interactive process waiting for a user to type something at a terminal is in I/O wait state, it should be clear that I/O wait times of 80 percent and more are not unusual. But even in batch systems, processes doing a lot of disk I/O will often have this percentage or more.

For the sake of complete accuracy, it should be pointed out that the probabilistic model just described is only an approximation. It implicitly assumes that all n processes are independent, meaning that it is quite acceptable for a system with five processes in memory to have three running and two waiting. But with a single CPU, we cannot have three processes running at once, so a process becoming ready while the CPU is busy will have to wait. Thus the processes are not independent. A more accurate model can be constructed using queueing theory, but the point we are making—multiprogramming lets processes use the CPU when it would be otherwise idle—is, of course, still valid, even if the true curves of Fig. 4-3 are slightly different.

Even though the model of Fig. 4-3 is simple-minded, it can nevertheless be used to make specific, although approximate, predictions about CPU performance.

Suppose, for example, that a computer has 32 MB of memory, with the operating system taking up 16 MB and each user program taking up 4 MB. These sizes allow four user programs to be in memory at once. With an 80 percent average I/O wait, we have a CPU utilization (ignoring operating system overhead) of $1 - 0.8^4$ or about 60 percent. Adding another 16 MB of memory allows the system to go from four-way multiprogramming to eight-way multiprogramming, thus raising the CPU utilization to 83 percent. In other words, the additional 16 MB will raise the throughput by 38 percent.

Adding yet another 16 MB would only increase CPU utilization from 83 percent to 93 percent, thus raising the throughput by only another 12 percent. Using this model the computer's owner might decide that the first addition is a good investment but that the second is not.

4.1.4 Analysis of Multiprogramming System Performance

The model discussed above can also be used to analyze batch systems. Consider, for example, a computer center whose jobs average 80 percent I/O wait time. On a particular morning, four jobs are submitted as shown in Fig. 4-4(a). The first job, arriving at 10:00 A.M., requires 4 minutes of CPU time. With 80 percent I/O wait, the job uses only 12 seconds of CPU time for each minute it is sitting in memory, even if no other jobs are competing with it for the CPU. The other 48 seconds are spent waiting for I/O to complete. Thus the job will have to sit in memory for at least 20 minutes in order to get 4 minutes of CPU work done, even in the absence of competition for the CPU.

From 10:00 A.M. to 10:10 A.M., job 1 is all by itself in memory and gets 2 minutes of work done. When job 2 arrives at 10:10 A.M., the CPU utilization increases from 0.20 to 0.36, due to the higher degree of multiprogramming (see Fig. 4-3). However, with round-robin scheduling, each job gets half of the CPU, so each job gets 0.18 minutes of CPU work done for each minute it is in memory. Notice that the addition of a second job costs the first job only 10 percent of its performance. It goes from getting 0.20 CPU minutes per minute of real time to getting 0.18 CPU minutes per minute of real time.

At 10:15 A.M. the third job arrives. At this point job 1 has received 2.9 minutes of CPU and job 2 has had 0.9 minutes of CPU. With three-way multiprogramming, each job gets 0.16 minutes of CPU time per minute of real time, as shown in Fig. 4-4(b). From 10:15 A.M. to 10:20 A.M. each of the three jobs gets 0.8 minutes of CPU time. At 10:20 A.M. a fourth job arrives. Fig. 4-4(c) shows the complete sequence of events.

4.1.5 Relocation and Protection

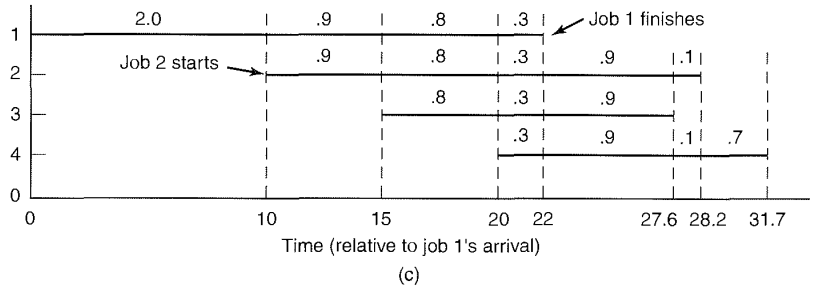
Multiprogramming introduces two essential problems that must be solved—relocation and protection. Look at Fig. 4-2. From the figure it is clear that different jobs will be run at different addresses. When a program is linked (i.e., the

Job	Arrival time	CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

(a)

	# Processes 1	2	3	4
CPU idle	.80	.64	.51	.41
CPU busy	.20	.36	.49	.59
CPU/process	.20	.18	.16	.15

(b)



(c)

Figure 4-4. (a) Arrival and work requirements of four jobs. (b) CPU utilization for 1 to 4 jobs with 80 percent I/O wait. (c) Sequence of events as jobs arrive and finish. The numbers above the horizontal lines show how much CPU time, in minutes, each job gets in each interval.

main program, user-written procedures, and library procedures are combined into a single address space), the linker must know at what address the program will begin in memory.

For example, suppose that the first instruction is a call to a procedure at absolute address 100 within the binary file produced by the linker. If this program is loaded in partition 1 (at address 100K), that instruction will jump to absolute address 100, which is inside the operating system. What is needed is a call to 100K + 100. If the program is loaded into partition 2, it must be carried out as a call to 200K + 100, and so on. This problem is known as the **relocation** problem.

One possible solution is to actually modify the instructions as the program is loaded into memory. Programs loaded into partition 1 have 100K added to each address, programs loaded into partition 2 have 200K added to addresses, and so forth. To perform relocation during loading like this, the linker must include in the binary program a list or bitmap telling which program words are addresses to be relocated and which are opcodes, constants, or other items that must not be relocated. OS/MFT worked this way.

Relocation during loading does not solve the protection problem. A malicious program can always construct a new instruction and jump to it. Because programs in this system use absolute memory addresses rather than addresses relative to a

register, there is no way to stop a program from building an instruction that reads or writes any word in memory. In multiuser systems, it is highly undesirable to let processes read and write memory belonging to other users.

The solution that IBM chose for protecting the 360 was to divide memory into blocks of 2-KB bytes and assign a 4-bit protection code to each block. The PSW (Program Status Word) contained a 4-bit key. The 360 hardware trapped any attempt by a running process to access memory whose protection code differed from the PSW key. Since only the operating system could change the protection codes and key, user processes were prevented from interfering with one another and with the operating system itself.

An alternative solution to both the relocation and protection problems is to equip the machine with two special hardware registers, called the **base** and **limit** registers. When a process is scheduled, the base register is loaded with the address of the start of its partition, and the limit register is loaded with the length of the partition. Every memory address generated automatically has the base register contents added to it before being sent to memory. Thus if the base register contains the value 100K, a CALL 100 instruction is effectively turned into a CALL 100K + 100 instruction, without the instruction itself being modified. Addresses are also checked against the limit register to make sure that they do not attempt to address memory outside the current partition. The hardware protects the base and limit registers to prevent user programs from modifying them.

A disadvantage of this scheme is the need to perform an addition and a comparison on every memory reference. Comparisons can be done fast, but additions are slow due to carry propagation time unless special addition circuits are used.

The CDC 6600—the world's first supercomputer—used this scheme. The Intel 8088 CPU used for the original IBM PC used a weaker version of this scheme—base registers, but no limit registers. Few computers use it any more though.

4.2 SWAPPING

With a batch system, organizing memory into fixed partitions is simple and effective. Each job is loaded into a partition when it gets to the head of the queue. It stays in memory until it has finished. As long as enough jobs can be kept in memory to keep the CPU busy all the time, there is no reason to use anything more complicated.

With timesharing systems or graphically oriented personal computers, the situation is different. Sometimes there is not enough main memory to hold all the currently active processes, so excess processes must be kept on disk and brought in to run dynamically.

Two general approaches to memory management can be used, depending (in part) on the available hardware. The simplest strategy, called **swapping**, consists

of bringing in each process in its entirety, running it for a while, then putting it back on the disk. The other strategy, called **virtual memory**, allows programs to run even when they are only partially in main memory. Below we will study swapping; in Sec. 4.3 we will examine virtual memory.

The operation of a swapping system is illustrated in Fig. 4-5. Initially only process *A* is in memory. Then processes *B* and *C* are created or swapped in from disk. In Fig. 4-5(d) *A* is swapped out to disk. Then *D* comes in and *B* goes out. Finally *A* comes in again. Since *A* is now at a different location, addresses contained in it must be relocated, either by software when it is swapped in or (more likely) by hardware during program execution.

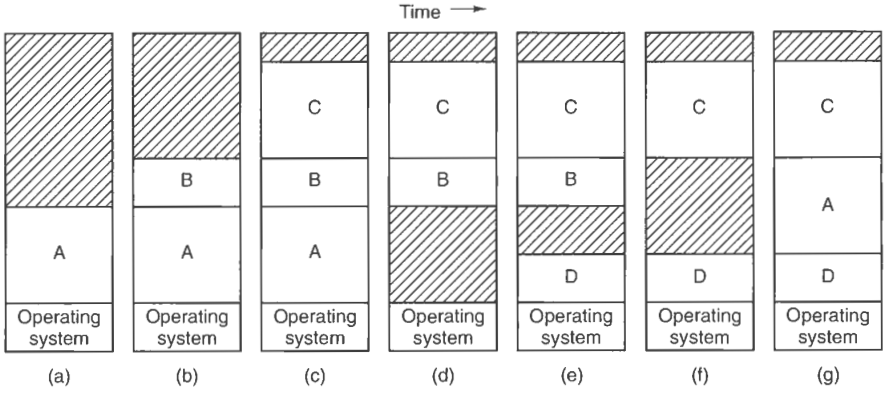


Figure 4-5. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

The main difference between the fixed partitions of Fig. 4-2 and the variable partitions of Fig. 4-5 is that the number, location, and size of the partitions vary dynamically in the latter as processes come and go, whereas they are fixed in the former. The flexibility of not being tied to a fixed number of partitions that may be too large or too small improves memory utilization, but it also complicates allocating and deallocating memory, as well as keeping track of it.

When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible. This technique is known as **memory compaction**. It is usually not done because it requires a lot of CPU time. For example, on a 256-MB machine that can copy 4 bytes in 40 nsec, it takes about 2.7 sec to compact all of memory.

A point that is worth making concerns how much memory should be allocated for a process when it is created or swapped in. If processes are created with a fixed size that never changes, then the allocation is simple: the operating system allocates exactly what is needed, no more and no less.

If, however, processes' data segments can grow, for example, by dynamically allocating memory from a heap, as in many programming languages, a problem occurs whenever a process tries to grow. If a hole is adjacent to the process, it can be allocated and the process allowed to grow into the hole. On the other hand, if the process is adjacent to another process, the growing process will either have to be moved to a hole in memory large enough for it, or one or more processes will have to be swapped out to create a large enough hole. If a process cannot grow in memory and the swap area on the disk is full, the process will have to wait or be killed.

If it is expected that most processes will grow as they run, it is probably a good idea to allocate a little extra memory whenever a process is swapped in or moved, to reduce the overhead associated with moving or swapping processes that no longer fit in their allocated memory. However, when swapping processes to disk, only the memory actually in use should be swapped; it is wasteful to swap the extra memory as well. In Fig. 4-6(a) we see a memory configuration in which space for growth has been allocated to two processes.

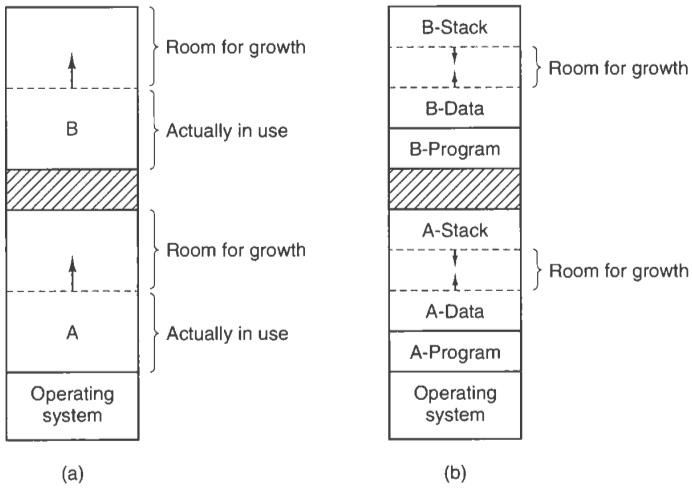


Figure 4-6. (a) Allocating space for a growing data segment. (b) Allocating space for a growing stack and a growing data segment.

If processes can have two growing segments, for example, the data segment being used as a heap for variables that are dynamically allocated and released and a stack segment for the normal local variables and return addresses, an alternative arrangement suggests itself, namely that of Fig. 4-6(b). In this figure we see that each process illustrated has a stack at the top of its allocated memory that is growing downward, and a data segment just beyond the program text that is growing

upward. The memory between them can be used for either segment. If it runs out, either the process will have to be moved to a hole with enough space, swapped out of memory until a large enough hole can be created, or killed.

4.2.1 Memory Management with Bitmaps

When memory is assigned dynamically, the operating system must manage it. In general terms, there are two ways to keep track of memory usage: bitmaps and free lists. In this section and the next one we will look at these two methods in turn.

With a bitmap, memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa). Figure 4-7 shows part of memory and the corresponding bitmap.

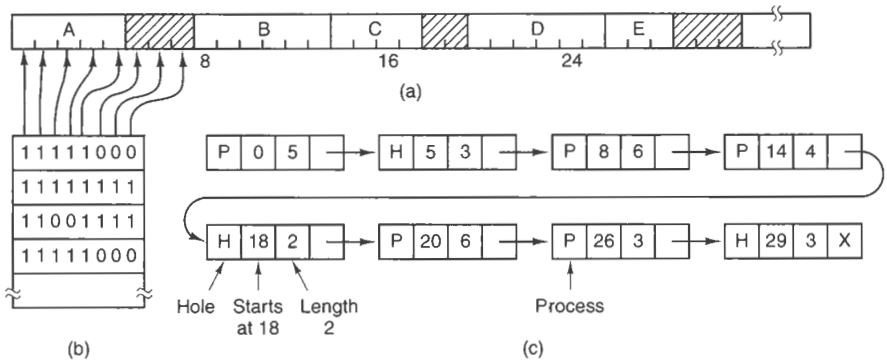


Figure 4-7. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

The size of the allocation unit is an important design issue. The smaller the allocation unit, the larger the bitmap. However, even with an allocation unit as small as 4 bytes, 32 bits of memory will require only 1 bit of the map. A memory of $32n$ bits will use n map bits, so the bitmap will take up only 1/33 of memory. If the allocation unit is chosen large, the bitmap will be smaller, but appreciable memory may be wasted in the last unit of the process if the process size is not an exact multiple of the allocation unit.

A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit. The main problem with it is that when

it has been decided to bring a k unit process into memory, the memory manager must search the bitmap to find a run of k consecutive 0 bits in the map. Searching a bitmap for a run of a given length is a slow operation (because the run may straddle word boundaries in the map); this is an argument against bitmaps.

4.2.2 Memory Management with Linked Lists

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes. The memory of Fig. 4-7(a) is represented in Fig. 4-7(c) as a linked list of segments. Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry.

In this example, the segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward. A terminating process normally has two neighbors (except when it is at the very top or bottom of memory). These may be either processes or holes, leading to the four combinations of Fig. 4-8. In Fig. 4-8(a) updating the list requires replacing a P by an H. In Fig. 4-8(b) and Fig. 4-8(c), two entries are coalesced into one, and the list becomes one entry shorter. In Fig. 4-8(d), three entries are merged and two items are removed from the list. Since the process table slot for the terminating process will normally point to the list entry for the process itself, it may be more convenient to have the list as a double-linked list, rather than the single-linked list of Fig. 4-7(c). This structure makes it easier to find the previous entry and to see if a merge is possible.

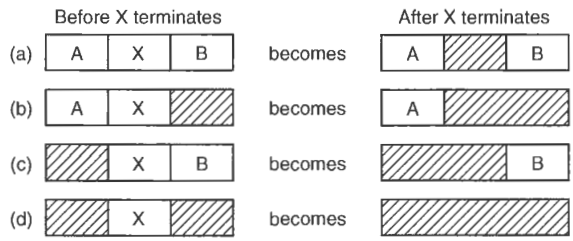


Figure 4-8. Four neighbor combinations for the terminating process, X.

When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created process (or an existing process being swapped in from disk). We assume that the memory manager knows how much memory to allocate. The simplest algorithm is **first fit**. The memory manager scans along the list of segments until it finds a hole that is big enough. The hole is then broken up into two pieces, one for the process and one for the unused memory, except in the statistically unlikely case of an exact fit. First fit is a fast algorithm because it searches as little as possible.

A minor variation of first fit is **next fit**. It works the same way as first fit, except that it keeps track of where it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at the beginning, as first fit does. Simulations by Bays (1977) show that next fit gives slightly worse performance than first fit.

Another well-known algorithm is **best fit**. Best fit searches the entire list and takes the smallest hole that is adequate. Rather than breaking up a big hole that might be needed later, best fit tries to find a hole that is close to the actual size needed.

As an example of first fit and best fit, consider Fig. 4-7 again. If a block of size 2 is needed, first fit will allocate the hole at 5, but best fit will allocate the hole at 18.

Best fit is slower than first fit because it must search the entire list every time it is called. Somewhat surprisingly, it also results in more wasted memory than first fit or next fit because it tends to fill up memory with tiny, useless holes. First fit generates larger holes on the average.

To get around the problem of breaking up nearly exact matches into a process and a tiny hole, one could think about **worst fit**, that is, always take the largest available hole, so that the hole broken off will be big enough to be useful. Simulation has shown that worst fit is not a very good idea either.

All four algorithms can be speeded up by maintaining separate lists for processes and holes. In this way, all of them devote their full energy to inspecting holes, not processes. The inevitable price that is paid for this speedup on allocation is the additional complexity and slowdown when deallocating memory, since a freed segment has to be removed from the process list and inserted into the hole list.

If distinct lists are maintained for processes and holes, the hole list may be kept sorted on size, to make best fit faster. When best fit searches a list of holes from smallest to largest, as soon as it finds a hole that fits, it knows that the hole is the smallest one that will do the job, hence the best fit. No further searching is needed, as it is with the single list scheme. With a hole list sorted by size, first fit and best fit are equally fast, and next fit is pointless.

When the holes are kept on separate lists from the processes, a small optimization is possible. Instead of having a separate set of data structures for maintaining the hole list, as is done in Fig. 4-7(c), the holes themselves can be used. The first word of each hole could be the hole size, and the second word a pointer to the following entry. The nodes of the list of Fig. 4-7(c), which require three words and one bit (P/H), are no longer needed.

Yet another allocation algorithm is **quick fit**, which maintains separate lists for some of the more common sizes requested. For example, it might have a table with n entries, in which the first entry is a pointer to the head of a list of 4-KB holes, the second entry is a pointer to a list of 8-KB holes, the third entry a pointer to 12-KB holes, and so on. Holes of say, 21 KB, could either be put on the 20-KB

list or on a special list of odd-sized holes. With quick fit, finding a hole of the required size is extremely fast, but it has the same disadvantage as all schemes that sort by hole size, namely, when a process terminates or is swapped out, finding its neighbors to see if a merge is possible is expensive. If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.

4.3 VIRTUAL MEMORY

Many years ago people were first confronted with programs that were too big to fit in the available memory. The solution usually adopted was to split the program into pieces, called **overlays**. Overlay 0 would start running first. When it was done, it would call another overlay. Some overlay systems were highly complex, allowing multiple overlays in memory at once. The overlays were kept on the disk and swapped in and out of memory by the operating system, dynamically, as needed.

Although the actual work of swapping overlays in and out was done by the system, the work of splitting the program into pieces had to be done by the programmer. Splitting up large programs into small, modular pieces was time consuming and boring. It did not take long before someone thought of a way to turn the whole job over to the computer.

The method that was devised (Fotheringham, 1961) has come to be known as **virtual memory**. The basic idea behind virtual memory is that the combined size of the program, data, and stack may exceed the amount of physical memory available for it. The operating system keeps those parts of the program currently in use in main memory, and the rest on the disk. For example, a 16-MB program can run on a 4-MB machine by carefully choosing which 4 MB to keep in memory at each instant, with pieces of the program being swapped between disk and memory as needed.

Virtual memory can also work in a multiprogramming system, with bits and pieces of many programs in memory at once. While a program is waiting for part of itself to be brought in, it is waiting for I/O and cannot run, so the CPU can be given to another process, the same way as in any other multiprogramming system.

4.3.1 Paging

Most virtual memory systems use a technique called **paging**, which we will now describe. On any computer, there exists a set of memory addresses that programs can produce. When a program uses an instruction like

```
MOV REG,1000
```

it does this to copy the contents of memory address 1000 to REG (or vice versa,

depending on the computer). Addresses can be generated using indexing, base registers, segment registers, and other ways.

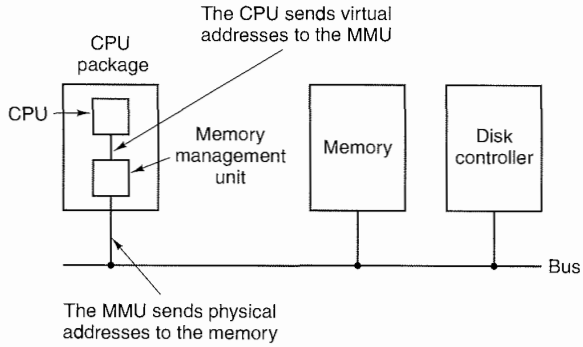


Figure 4-9. The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays. However, logically it could be a separate chip and was in years gone by.

These program-generated addresses are called **virtual addresses** and form the **virtual address space**. On computers without virtual memory, the virtual address is put directly onto the memory bus and causes the physical memory word with the same address to be read or written. When virtual memory is used, the virtual addresses do not go directly to the memory bus. Instead, they go to an **MMU** (**Memory Management Unit**) that maps the virtual addresses onto the physical memory addresses as illustrated in Fig. 4-9.

A very simple example of how this mapping works is shown in Fig. 4-10. In this example, we have a computer that can generate 16-bit addresses, from 0 up to 64K. These are the virtual addresses. This computer, however, has only 32 KB of physical memory, so although 64-KB programs can be written, they cannot be loaded into memory in their entirety and run. A complete copy of a program's core image, up to 64 KB, must be present on the disk, however, so that pieces can be brought in as needed.

The virtual address space is divided up into units called **pages**. The corresponding units in the physical memory are called **page frames**. The pages and page frames are always the same size. In this example they are 4 KB, but page sizes from 512 bytes to 64 KB have been used in real systems. With 64 KB of virtual address space and 32 KB of physical memory, we get 16 virtual pages and 8 page frames. Transfers between RAM and disk are always in units of a page.

When the program tries to access address 0, for example, using the instruction

```
MOV REG,0
```

virtual address 0 is sent to the MMU. The MMU sees that this virtual address falls

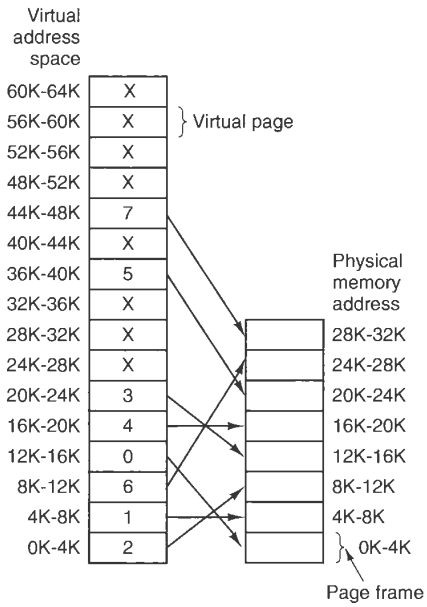


Figure 4-10. The relation between virtual addresses and physical memory addresses is given by the page table.

in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287). It thus transforms the address to 8192 and outputs address 8192 onto the bus. The memory knows nothing at all about the MMU and just sees a request for reading or writing address 8192, which it honors. Thus, the MMU has effectively mapped all virtual addresses between 0 and 4095 onto physical addresses 8192 to 12287.

Similarly, an instruction

```
MOV REG,8192
```

is effectively transformed into

```
MOV REG,24576
```

because virtual address 8192 is in virtual page 2 and this page is mapped onto physical page frame 6 (physical addresses 24576 to 28671). As a third example, virtual address 20500 is 20 bytes from the start of virtual page 5 (virtual addresses 20480 to 24575) and maps onto physical address 12288 + 20 = 12308.

By itself, this ability to map the 16 virtual pages onto any of the eight page frames by setting the MMU's map appropriately does not solve the problem that the virtual address space is larger than the physical memory. Since we have only

eight physical page frames, only eight of the virtual pages in Fig. 4-10 are mapped onto physical memory. The others, shown as a cross in the figure, are not mapped. In the actual hardware, a **Present/absent bit** keeps track of which pages are physically present in memory.

What happens if the program tries to use an unmapped page, for example, by using the instruction

```
MOV REG,32780
```

which is byte 12 within virtual page 8 (starting at 32768)? The MMU notices that the page is unmapped (indicated by a cross in the figure) and causes the CPU to trap to the operating system. This trap is called a **page fault**. The operating system picks a little-used page frame and writes its contents back to the disk. It then fetches the page just referenced into the page frame just freed, changes the map, and restarts the trapped instruction.

For example, if the operating system decided to evict page frame 1, it would load virtual page 8 at physical address 4K and make two changes to the MMU map. First, it would mark virtual page 1's entry as unmapped, to trap any future accesses to virtual addresses between 4K and 8K. Then it would replace the cross in virtual page 8's entry with a 1, so that when the trapped instruction is re-executed, it will map virtual address 32780 onto physical address 4108.

Now let us look inside the MMU to see how it works and why we have chosen to use a page size that is a power of 2. In Fig. 4-11 we see an example of a virtual address, 8196 (0010000000000100 in binary), being mapped using the MMU map of Fig. 4-10. The incoming 16-bit virtual address is split into a 4-bit page number and a 12-bit offset. With 4 bits for the page number, we can have 16 pages, and with 12 bits for the offset, we can address all 4096 bytes within a page.

The page number is used as an index into the **page table**, yielding the number of the page frame corresponding to that virtual page. If the *Present/absent* bit is 0, a trap to the operating system is caused. If the bit is 1, the page frame number found in the page table is copied to the high-order 3 bits of the output register, along with the 12-bit offset, which is copied unmodified from the incoming virtual address. Together they form a 15-bit physical address. The output register is then put onto the memory bus as the physical memory address.

4.3.2 Page Tables

In the simplest case, the mapping of virtual addresses onto physical addresses is as we have just described it. The virtual address is split into a virtual page number (high-order bits) and an offset (low-order bits). For example, with a 16-bit address and a 4-KB page size, the upper 4 bits could specify one of the 16 virtual pages and the lower 12 bits would then specify the byte offset (0 to 4095) within the selected page. However a split with 3 or 5 or some other number of bits for the page is also possible. Different splits imply different page sizes.

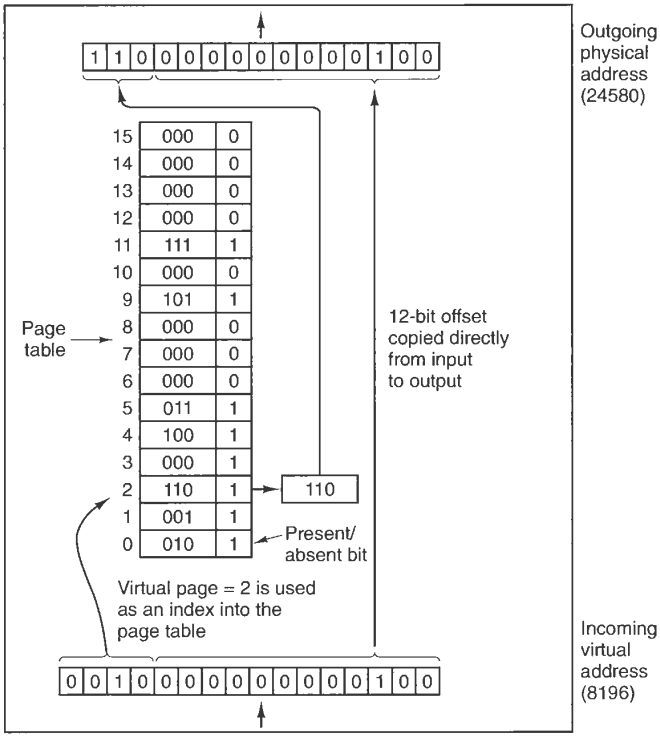


Figure 4-11. The internal operation of the MMU with 16 4-KB pages.

The virtual page number is used as an index into the page table to find the entry for that virtual page. From the page table entry, the page frame number (if any) is found. The page frame number is attached to the high-order end of the offset, replacing the virtual page number, to form a physical address that can be sent to the memory.

The purpose of the page table is to map virtual pages onto page frames. Mathematically speaking, the page table is a function, with the virtual page number as argument and the physical frame number as result. Using the result of this function, the virtual page field in a virtual address can be replaced by a page frame field, thus forming a physical memory address.

Despite this simple description, two major issues must be faced:

1. The page table can be extremely large.
2. The mapping must be fast.

The first point follows from the fact that modern computers use virtual addresses of at least 32 bits. With, say, a 4-KB page size, a 32-bit address space has 1 million pages, and a 64-bit address space has more than you want to contemplate. With 1 million pages in the virtual address space, the page table must have 1 million entries. And remember that each process needs its own page table (because it has its own virtual address space).

The second point is a consequence of the fact that the virtual-to-physical mapping must be done on every memory reference. A typical instruction has an instruction word, and often a memory operand as well. Consequently, it is necessary to make 1, 2, or sometimes more page table references per instruction. If an instruction takes, say, 4 nsec, the page table lookup must be done in under 1 nsec to avoid becoming a major bottleneck.

The need for large, fast page mapping is a significant constraint on the way computers are built. Although the problem is most serious with top-of-the-line machines, it is also an issue at the low end as well, where cost and the price/performance ratio are critical. In this section and the following ones, we will look at page table design in detail and show a number of hardware solutions that have been used in actual computers.

The simplest design (at least conceptually) is to have a single page table consisting of an array of fast hardware registers, with one entry for each virtual page, indexed by virtual page number, as shown in Fig. 4-11. When a process is started up, the operating system loads the registers with the process' page table, taken from a copy kept in main memory. During process execution, no more memory references are needed for the page table. The advantages of this method are that it is straightforward and requires no memory references during mapping. A disadvantage is that it is potentially expensive (if the page table is large). Having to load the full page table at every context switch hurts performance.

At the other extreme, the page table can be entirely in main memory. All the hardware needs then is a single register that points to the start of the page table. This design allows the memory map to be changed at a context switch by reloading one register. Of course, it has the disadvantage of requiring one or more memory references to read page table entries during the execution of each instruction. For this reason, this approach is rarely used in its most pure form, but below we will study some variations that have much better performance.

Multilevel Page Tables

To get around the problem of having to store huge page tables in memory all the time, many computers use a multilevel page table. A simple example is shown in Fig. 4-12. In Fig. 4-12(a) we have a 32-bit virtual address that is partitioned into a 10-bit *PT1* field, a 10-bit *PT2* field, and a 12-bit *Offset* field. Since offsets are 12 bits, pages are 4 KB, and there are a total of 2^{20} of them.

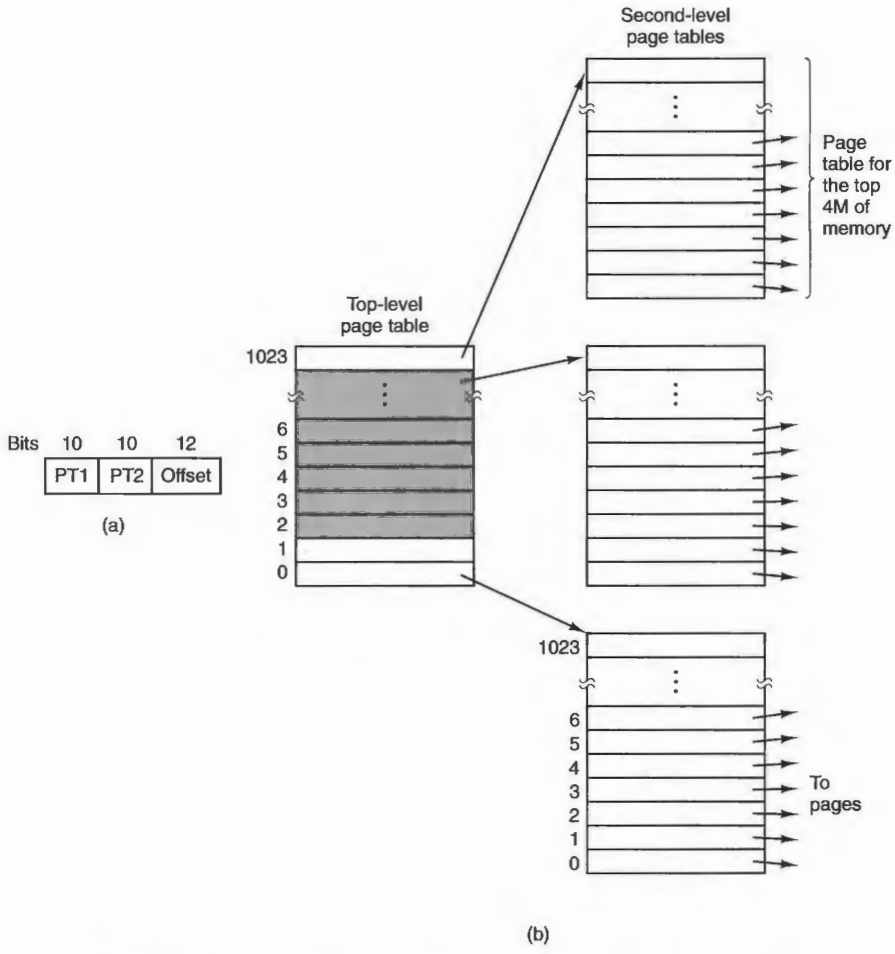


Figure 4-12. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time. In particular, those that are not needed should not be kept around. Suppose, for example, that a process needs 12 megabytes, the bottom 4 megabytes of memory for program text, the next 4 megabytes for data, and the top 4 megabytes for the stack. In between the top of the data and the bottom of the stack is a gigantic hole that is not used.

In Fig. 4-12(b) we see how the two-level page table works in this example. On the left we have the top-level page table, with 1024 entries, corresponding to

the 10-bit *PT1* field. When a virtual address is presented to the MMU, it first extracts the *PT1* field and uses this value as an index into the top-level page table. Each of these 1024 entries represents 4M because the entire 4-gigabyte (i.e., 32-bit) virtual address space has been chopped into chunks of 1024 bytes.

The entry located by indexing into the top-level page table yields the address or the page frame number of a second-level page table. Entry 0 of the top-level page table points to the page table for the program text, entry 1 points to the page table for the data, and entry 1023 points to the page table for the stack. The other (shaded) entries are not used. The *PT2* field is now used as an index into the selected second-level page table to find the page frame number for the page itself.

As an example, consider the 32-bit virtual address 0x00403004 (4,206,596 decimal), which is 12,292 bytes into the data. This virtual address corresponds to $PT1 = 1$, $PT2 = 2$, and $Offset = 4$. The MMU first uses *PT1* to index into the top-level page table and obtain entry 1, which corresponds to addresses 4M to 8M. It then uses *PT2* to index into the second-level page table just found and extract entry 3, which corresponds to addresses 12288 to 16383 within its 4M chunk (i.e., absolute addresses 4,206,592 to 4,210,687). This entry contains the page frame number of the page containing virtual address 0x00403004. If that page is not in memory, the *Present/absent* bit in the page table entry will be zero, causing a page fault. If the page is in memory, the page frame number taken from the second-level page table is combined with the offset (4) to construct a physical address. This address is put on the bus and sent to memory.

The interesting thing to note about Fig. 4-12 is that although the address space contains over a million pages, only four page tables are actually needed: the top-level table, and the second-level tables for 0 to 4M, 4M to 8M, and the top 4M. The *Present/absent* bits in 1021 entries of the top-level page table are set to 0, forcing a page fault if they are ever accessed. Should this occur, the operating system will notice that the process is trying to reference memory that it is not supposed to and will take appropriate action, such as sending it a signal or killing it. In this example we have chosen round numbers for the various sizes and have picked *PT1* equal to *PT2* but in actual practice other values are also possible, of course.

The two-level page table system of Fig. 4-12 can be expanded to three, four, or more levels. Additional levels give more flexibility, but it is doubtful that the additional complexity is worth it beyond three levels.

Structure of a Page Table Entry

Let us now turn from the structure of the page tables in the large, to the details of a single page table entry. The exact layout of an entry is highly machine dependent, but the kind of information present is roughly the same from machine to machine. In Fig. 4-13 we give a sample page table entry. The size varies from computer to computer, but 32 bits is a common size. The most important field is

the *Page frame number*. After all, the goal of the page mapping is to locate this value. Next to it we have the *Present/absent* bit. If this bit is 1, the entry is valid and can be used. If it is 0, the virtual page to which the entry belongs is not currently in memory. Accessing a page table entry with this bit set to 0 causes a page fault.

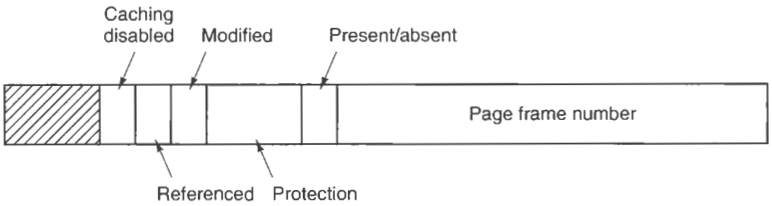


Figure 4-13. A typical page table entry.

The *Protection* bits tell what kinds of access are permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only. A more sophisticated arrangement is having 3 bits, one bit each for enabling reading, writing, and executing the page.

The *Modified* and *Referenced* bits keep track of page usage. When a page is written to, the hardware automatically sets the *Modified* bit. This bit is of value when the operating system decides to reclaim a page frame. If the page in it has been modified (i.e., is "dirty"), it must be written back to the disk. If it has not been modified (i.e., is "clean"), it can just be abandoned, since the disk copy is still valid. The bit is sometimes called the **dirty bit**, since it reflects the page's state.

The *Referenced* bit is set whenever a page is referenced, either for reading or writing. Its value is to help the operating system choose a page to evict when a page fault occurs. Pages that are not being used are better candidates than pages that are, and this bit plays an important role in several of the page replacement algorithms that we will study later in this chapter.

Finally, the last bit allows caching to be disabled for the page. This feature is important for pages that map onto device registers rather than memory. If the operating system is sitting in a tight loop waiting for some I/O device to respond to a command it was just given, it is essential that the hardware keep fetching the word from the device, and not use an old cached copy. With this bit, caching can be turned off. Machines that have a separate I/O space and do not use memory mapped I/O do not need this bit.

Note that the disk address used to hold the page when it is not in memory is not part of the page table. The reason is simple. The page table holds only that information the hardware needs to translate a virtual address to a physical address. Information the operating system needs to handle page faults is kept in software tables inside the operating system. The hardware does not need it.

4.3.3 TLBs—Translation Lookaside Buffers

In most paging schemes, the page tables are kept in memory, due to their large size. Potentially, this design has an enormous impact on performance. Consider, for example, an instruction that copies one register to another. In the absence of paging, this instruction makes only one memory reference, to fetch the instruction. With paging, additional memory references will be needed to access the page table. Since execution speed is generally limited by the rate the CPU can get instructions and data out of the memory, having to make two page table references per memory reference reduces performance by 2/3. Under these conditions, no one would use it.

Computer designers have known about this problem for years and have come up with a solution. Their solution is based on the observation that most programs tend to make a large number of references to a small number of pages, and not the other way around. Thus only a small fraction of the page table entries are heavily read; the rest are barely used at all.

The solution that has been devised is to equip computers with a small hardware device for mapping virtual addresses to physical addresses without going through the page table. The device, called a **TLB (Translation Lookaside Buffer)** or sometimes an **associative memory**, is illustrated in Fig. 4-14. It is usually inside the MMU and consists of a small number of entries, eight in this example, but rarely more than 64. Each entry contains information about one page, including the virtual page number, a bit that is set when the page is modified, the protection code (read/write/execute permissions), and the physical page frame in which the page is located. These fields have a one-to-one correspondence with the fields in the page table. Another bit indicates whether the entry is valid (i.e., in use) or not.

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 4-14. A TLB to speed up paging.

An example that might generate the TLB of Fig. 4-14 is a process in a loop that spans virtual pages 19, 20, and 21, so these TLB entries have protection codes

for reading and executing. The main data currently being used (say, an array being processed) are on pages 129 and 130. Page 140 contains the indices used in the array calculations. Finally, the stack is on pages 860 and 861.

Let us now see how the TLB functions. When a virtual address is presented to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB by comparing it to all the entries simultaneously (i.e., in parallel). If a valid match is found and the access does not violate the protection bits, the page frame is taken directly from the TLB, without going to the page table. If the virtual page number is present in the TLB but the instruction is trying to write on a read-only page, a protection fault is generated, the same way as it would be from the page table itself.

The interesting case is what happens when the virtual page number is not in the TLB. The MMU detects the miss and does an ordinary page table lookup. It then evicts one of the entries from the TLB and replaces it with the page table entry just looked up. Thus if that page is used again soon, the second time it will result in a hit rather than a miss. When an entry is purged from the TLB, the modified bit is copied back into the page table entry in memory. The other values are already there. When the TLB is loaded from the page table, all the fields are taken from memory.

Software TLB Management

Up until now, we have assumed that every machine with paged virtual memory has page tables recognized by the hardware, plus a TLB. In this design, TLB management and handling TLB faults are done entirely by the MMU hardware. Traps to the operating system occur only when a page is not in memory.

In the past, this assumption was true. However, many modern RISC machines, including the SPARC, MIPS, Alpha, and HP PA, do nearly all of this page management in software. On these machines, the TLB entries are explicitly loaded by the operating system. When a TLB miss occurs, instead of the MMU just going to the page tables to find and fetch the needed page reference, it just generates a TLB fault and tosses the problem into the lap of the operating system. The system must find the page, remove an entry from the TLB, enter the new one, and restart the instruction that faulted. And, of course, all of this must be done in a handful of instructions because TLB misses occur much more frequently than page faults.

Surprisingly enough, if the TLB is reasonably large (say, 64 entries) to reduce the miss rate, software management of the TLB turns out to be acceptably efficient. The main gain here is a much simpler MMU, which frees up a considerable amount of area on the CPU chip for caches and other features that can improve performance. Software TLB management is discussed by Uhlig et al. (1994).

Various strategies have been developed to improve performance on machines that do TLB management in software. One approach attacks both reducing TLB

misses and reducing the cost of a TLB miss when it does occur (Bala et al., 1994). To reduce TLB misses, sometimes the operating system can use its intuition to figure out which pages are likely to be used next and to preload entries for them in the TLB. For example, when a client process sends a message to a server process on the same machine, it is very likely that the server will have to run soon. Knowing this, while processing the trap to do the send, the system can also check to see where the server's code, data, and stack pages are and map them in before they can cause TLB faults.

The normal way to process a TLB miss, whether in hardware or in software, is to go to the page table and perform the indexing operations to locate the page referenced. The problem with doing this search in software is that the pages holding the page table may not be in the TLB, which will cause additional TLB faults during the processing. These faults can be reduced by maintaining a large (e.g., 4-KB) software cache of TLB entries in a fixed location whose page is always kept in the TLB. By first checking the software cache, the operating system can substantially reduce TLB misses.

4.3.4 Inverted Page Tables

Traditional page tables of the type described so far require one entry per virtual page, since they are indexed by virtual page number. If the address space consists of 2^{32} bytes, with 4096 bytes per page, then over 1 million page table entries are needed. As a bare minimum, the page table will have to be at least 4 megabytes. On larger systems, this size is probably doable.

However, as 64-bit computers become more common, the situation changes drastically. If the address space is now 2^{64} bytes, with 4-KB pages, we need a page table with 2^{52} entries. If each entry is 8 bytes, the table is over 30 million gigabytes. Tying up 30 million gigabytes just for the page table is not doable, not now and not for years to come, if ever. Consequently, a different solution is needed for 64-bit paged virtual address spaces.

One such solution is the **inverted page table**. In this design, there is one entry per page frame in real memory, rather than one entry per page of virtual address space. For example, with 64-bit virtual addresses, a 4-KB page, and 256 MB of RAM, an inverted page table only requires 65,536 entries. The entry keeps track of which (process, virtual page) is located in the page frame.

Although inverted page tables save vast amounts of space, at least when the virtual address space is much larger than the physical memory, they have a serious downside: virtual-to-physical translation becomes much harder. When process n references virtual page p, the hardware can no longer find the physical page by using p as an index into the page table. Instead, it must search the entire inverted page table for an entry (n, p). Furthermore, this search must be done on every memory reference, not just on page faults. Searching a 64K table on every memory reference is not the way to make your machine blindingly fast.

The way out of this dilemma is to use the TLB. If the TLB can hold all of the heavily used pages, translation can happen just as fast as with regular page tables. On a TLB miss, however, the inverted page table has to be searched in software. One feasible way to accomplish this search is to have a hash table hashed on the virtual address. All the virtual pages currently in memory that have the same hash value are chained together, as shown in Fig. 4-15. If the hash table has as many slots as the machine has physical pages, the average chain will be only one entry long, greatly speeding up the mapping. Once the page frame number has been found, the new (virtual, physical) pair is entered into the TLB.

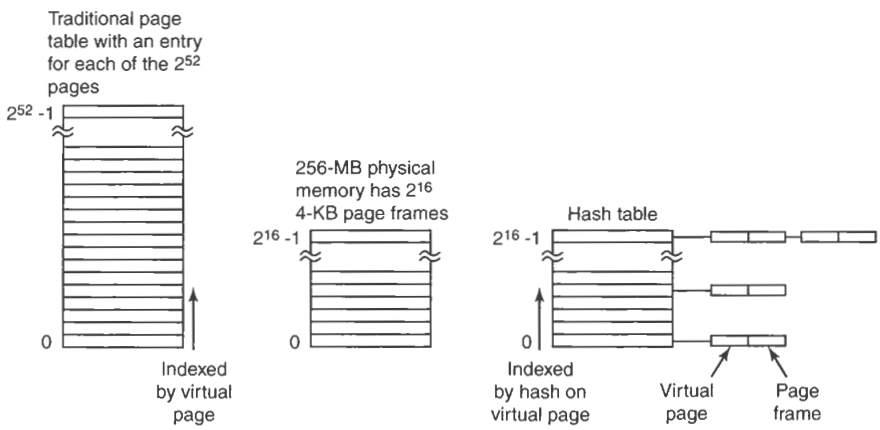


Figure 4-15. Comparison of a traditional page table with an inverted page table.

Inverted page tables are currently used on some IBM and Hewlett-Packard workstations and will become more common as 64-bit machines become widespread. Other approaches to handling large virtual memories can be found in (Huck and Hays, 1993; Talluri and Hill, 1994; and Talluri et al., 1995).

4.4 PAGE REPLACEMENT ALGORITHMS

When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. If the page to be removed has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed. The page to be read in just overwrites the page being evicted.

While it would be possible to pick a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen. If

a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental. Below we will describe some of the most important algorithms.

It is worth noting that the problem of "page replacement" occurs in other areas of computer design as well. For example, most computers have one or more memory caches consisting of recently used 32-byte or 64-byte memory blocks. When the cache is full, some block has to be chosen for removal. This problem is precisely the same as page replacement except on a shorter time scale (it has to be done in a few nanoseconds, not milliseconds as with page replacement). The reason for the shorter time scale is that cache block misses are satisfied from main memory, which has no seek time and no rotational latency.

A second example is in a Web server. The server can keep a certain number of heavily used Web pages in its memory cache. However, when the memory cache is full and a new page is referenced, a decision has to be made which Web page to evict. The considerations are similar to pages of virtual memory, except for the fact that the Web pages are never modified in the cache, so there is always a fresh copy on disk. In a virtual memory system, pages in main memory may be either clean or dirty.

4.4.1 The Optimal Page Replacement Algorithm

The best possible page replacement algorithm is easy to describe but impossible to implement. It goes like this. At the moment that a page fault occurs, some set of pages is in memory. One of these pages will be referenced on the very next instruction (the page containing that instruction). Other pages may not be referenced until 10, 100, or perhaps 1000 instructions later. Each page can be labeled with the number of instructions that will be executed before that page is first referenced.

The optimal page algorithm simply says that the page with the highest label should be removed. If one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, removing the former pushes the page fault that will fetch it back as far into the future as possible. Computers, like people, try to put off unpleasant events for as long as they can.

The only problem with this algorithm is that it is unrealizable. At the time of the page fault, the operating system has no way of knowing when each of the pages will be referenced next. (We saw a similar situation earlier with the shortest job first scheduling algorithm—how can the system tell which job is shortest?) Still, by running a program on a simulator and keeping track of all page references, it is possible to implement optimal page replacement on the *second* run by using the page reference information collected during the *first* run.

In this way it is possible to compare the performance of realizable algorithms with the best possible one. If an operating system achieves a performance of, say,

only 1 percent worse than the optimal algorithm, effort spent in looking for a better algorithm will yield at most a 1 percent improvement.

To avoid any possible confusion, it should be made clear that this log of page references refers only to the one program just measured and then with only one specific input. The page replacement algorithm derived from it is thus specific to that one program and input data. Although this method is useful for evaluating page replacement algorithms, it is of no use in practical systems. Below we will study algorithms that *are* useful on real systems.

4.4.2 The Not Recently Used Page Replacement Algorithm

In order to allow the operating system to collect useful statistics about which pages are being used and which ones are not, most computers with virtual memory have two status bits associated with each page. *R* is set whenever the page is referenced (read or written). *M* is set when the page is written to (i.e., modified). The bits are contained in each page table entry, as shown in Fig. 4-13. It is important to realize that these bits must be updated on every memory reference, so it is essential that they be set by the hardware. Once a bit has been set to 1, it stays 1 until the operating system resets it to 0 in software.

If the hardware does not have these bits, they can be simulated as follows. When a process is started up, all of its page table entries are marked as not in memory. As soon as any page is referenced, a page fault will occur. The operating system then sets the *R* bit (in its internal tables), changes the page table entry to point to the correct page, with mode READ ONLY, and restarts the instruction. If the page is subsequently written on, another page fault will occur, allowing the operating system to set the *M* bit and change the page's mode to READ/WRITE.

The *R* and *M* bits can be used to build a simple paging algorithm as follows. When a process is started up, both page bits for all its pages are set to 0 by the operating system. Periodically (e.g., on each clock interrupt), the *R* bit is cleared, to distinguish pages that have not been referenced recently from those that have been.

When a page fault occurs, the operating system inspects all the pages and divides them into four categories based on the current values of their *R* and *M* bits:

Class 0: not referenced, not modified.
Class 1: not referenced, modified.
Class 2: referenced, not modified.
Class 3: referenced, modified.

Although class 1 pages seem, at first glance, impossible, they occur when a class 3 page has its *R* bit cleared by a clock interrupt. Clock interrupts do not clear the *M* bit because this information is needed to know whether the page has to be rewritten to disk or not. Clearing *R* but not *M* leads to a class 1 page.

The **NRU (Not Recently Used)** algorithm removes a page at random from the lowest numbered nonempty class. Implicit in this algorithm is that it is better to remove a modified page that has not been referenced in at least one clock tick (typically 20 msec) than a clean page that is in heavy use. The main attraction of NRU is that it is easy to understand, moderately efficient to implement, and gives a performance that, while certainly not optimal, may be adequate.

4.4.3 The First-In, First-Out (FIFO) Page Replacement Algorithm

Another low-overhead paging algorithm is the **FIFO (First-In, First-Out)** algorithm. To illustrate how this works, consider a supermarket that has enough shelves to display exactly *k* different products. One day, some company introduces a new convenience food—instant, freeze-dried, organic yogurt that can be reconstituted in a microwave oven. It is an immediate success, so our finite supermarket has to get rid of one old product in order to stock it.

One possibility is to find the product that the supermarket has been stocking the longest (i.e., something it began selling 120 years ago) and get rid of it on the grounds that no one is interested any more. In effect, the supermarket maintains a linked list of all the products it currently sells in the order they were introduced. The new one goes on the back of the list; the one at the front of the list is dropped.

As a page replacement algorithm, the same idea is applicable. The operating system maintains a list of all pages currently in memory, with the page at the head of the list the oldest one and the page at the tail the most recent arrival. On a page fault, the page at the head is removed and the new page added to the tail of the list. When applied to stores, FIFO might remove mustache wax, but it might also remove flour, salt, or butter. When applied to computers the same problem arises. For this reason, FIFO in its pure form is rarely used.

4.4.4 The Second Chance Page Replacement Algorithm

A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the *R* bit of the oldest page. If it is 0, the page is both old and unused, so it is replaced immediately. If the *R* bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory. Then the search continues.

The operation of this algorithm, called **second chance**, is shown in Fig. 4-16. In Fig. 4-16(a) we see pages *A* through *H* kept on a linked list and sorted by the time they arrived in memory.

Suppose that a page fault occurs at time 20. The oldest page is *A*, which arrived at time 0, when the process started. If *A* has the *R* bit cleared, it is evicted from memory, either by being written to the disk (if it is dirty), or just abandoned (if it is clean). On the other hand, if the *R* bit is set, *A* is put onto the end of the

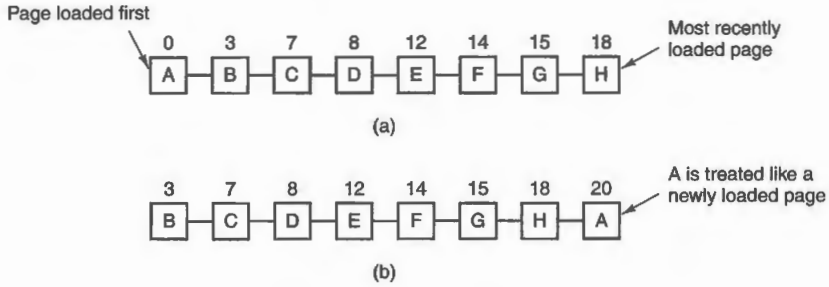


Figure 4-16. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their loading times.

list and its "load time" is reset to the current time (20). The *R* bit is also cleared. The search for a suitable page continues with *B*.

What second chance is doing is looking for an old page that has not been referenced in the previous clock interval. If all the pages have been referenced, second chance degenerates into pure FIFO. Specifically, imagine that all the pages in Fig. 4-16(a) have their *R* bits set. One by one, the operating system moves the pages to the end of the list, clearing the *R* bit each time it appends a page to the end of the list. Eventually, it comes back to page *A*, which now has its *R* bit cleared. At this point *A* is evicted. Thus the algorithm always terminates.

4.4.5 The Clock Page Replacement Algorithm

Although second chance is a reasonable algorithm, it is unnecessarily inefficient because it is constantly moving pages around on its list. A better approach is to keep all the page frames on a circular list in the form of a clock, as shown in Fig. 4-17. A hand points to the oldest page.

When a page fault occurs, the page being pointed to by the hand is inspected. If its *R* bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position. If *R* is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with $R = 0$. Not surprisingly, this algorithm is called **clock**. It differs from second chance only in the implementation.

4.4.6 The Least Recently Used (LRU) Page Replacement Algorithm

A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable

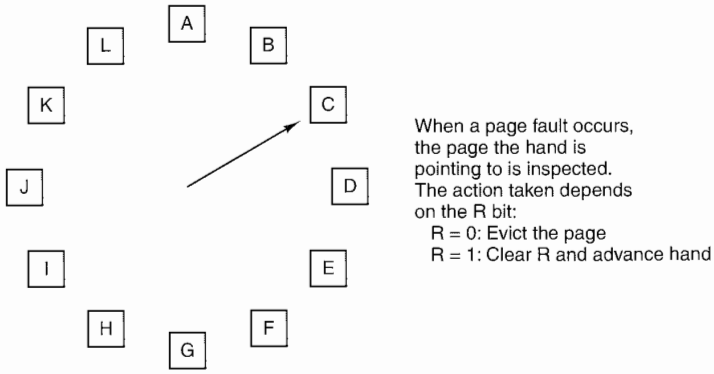


Figure 4-17. The clock page replacement algorithm.

algorithm: when a page fault occurs, throw out the page that has been unused for the longest time. This strategy is called **LRU (Least Recently Used)** paging.

Although LRU is theoretically realizable, it is not cheap. To fully implement LRU, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear. The difficulty is that the list must be updated on every memory reference. Finding a page in the list, deleting it, and then moving it to the front is a very time consuming operation, even in hardware (assuming that such hardware could be built).

However, there are other ways to implement LRU with special hardware. Let us consider the simplest way first. This method requires equipping the hardware with a 64-bit counter, C, that is automatically incremented after each instruction. Furthermore, each page table entry must also have a field large enough to contain the counter. After each memory reference, the current value of C is stored in the page table entry for the page just referenced. When a page fault occurs, the operating system examines all the counters in the page table to find the lowest one. That page is the least recently used.

Now let us look at a second hardware LRU algorithm. For a machine with n page frames, the LRU hardware can maintain a matrix of $n \times n$ bits, initially all zero. Whenever page frame k is referenced, the hardware first sets all the bits of row k to 1, then sets all the bits of column k to 0. At any instant, the row whose binary value is lowest is the least recently used, the row whose value is next lowest is next least recently used, and so forth. The workings of this algorithm are given in Fig. 4-18 for four page frames and page references in the order

0 1 2 3 2 1 0 3 2 3

After page 0 is referenced, we have the situation of Fig. 4-18(a). After page 1 is reference, we have the situation of Fig. 4-18(b), and so forth.

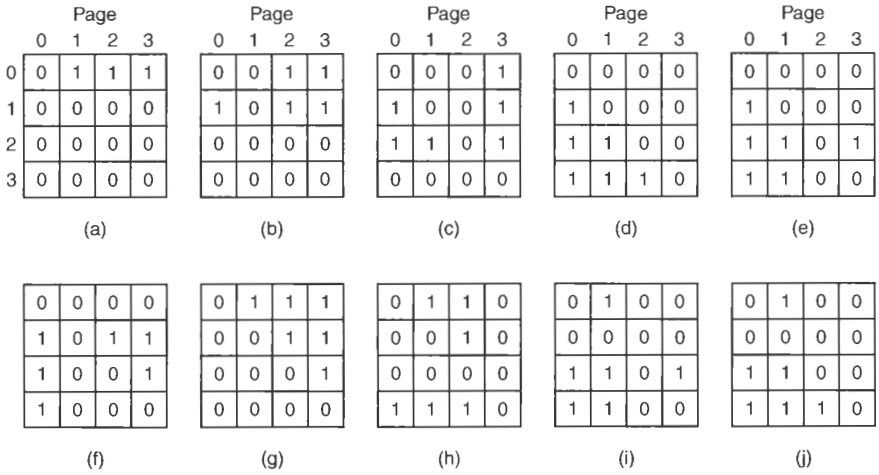


Figure 4-18. LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

4.4.7 Simulating LRU in Software

Although both of the previous LRU algorithms are realizable in principle, few, if any, machines have this hardware, so they are of little use to the operating system designer who is making a system for a machine that does not have this hardware. Instead, a solution that can be implemented in software is needed. One possibility is called the **NFU** (**Not Frequently Used**) algorithm. It requires a software counter associated with each page, initially zero. At each clock interrupt, the operating system scans all the pages in memory. For each page, the *R* bit, which is 0 or 1, is added to the counter. In effect, the counters are an attempt to keep track of how often each page has been referenced. When a page fault occurs, the page with the lowest counter is chosen for replacement.

The main problem with NFU is that it never forgets anything. For example, in a multipass compiler, pages that were heavily used during pass 1 may still have a high count well into later passes. In fact, if pass 1 happens to have the longest execution time of all the passes, the pages containing the code for subsequent passes may always have lower counts than the pass 1 pages. Consequently, the operating system will remove useful pages instead of pages no longer in use.

Fortunately, a small modification to NFU makes it able to simulate LRU quite well. The modification has two parts. First, the counters are each shifted right 1 bit before the *R* bit is added in. Second, the *R* bit is added to the leftmost, rather than the rightmost bit.

Figure 4-19 illustrates how the modified algorithm, known as **aging**, works. Suppose that after the first clock tick the *R* bits for pages 0 to 5 have the values 1,

0, 1, 0, 1, and 1, respectively (page 0 is 1, page 1 is 0, page 2 is 1, etc.). In other words, between tick 0 and tick 1, pages 0, 2, 4, and 5 were referenced, setting their *R* bits to 1, while the other ones remain 0. After the six corresponding counters have been shifted and the *R* bit inserted at the left, they have the values shown in Fig. 4-19(a). The four remaining columns show the six counters after the next four clock ticks.

	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
Page					
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00100000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000
	(a)	(b)	(c)	(d)	(e)

Figure 4-19. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

When a page fault occurs, the page whose counter is the lowest is removed. It is clear that a page that has not been referenced for, say, four clock ticks will have four leading zeros in its counter and thus will have a lower value than a counter that has not been referenced for three clock ticks.

This algorithm differs from LRU in two ways. Consider pages 3 and 5 in Fig. 4-19(e). Neither has been referenced for two clock ticks; both were referenced in the tick prior to that. According to LRU, if a page must be replaced, we should choose one of these two. The trouble is, we do not know which of these two was referenced last in the interval between tick 1 and tick 2. By recording only one bit per time interval, we have lost the ability to distinguish references early in the clock interval from those occurring later. All we can do is remove page 3, because page 5 was also referenced two ticks earlier and page 3 was not.

The second difference between LRU and aging is that in aging the counters have a finite number of bits, 8 bits in this example. Suppose that two pages each have a counter value of 0. All we can do is pick one of them at random. In

reality, it may well be that one of the pages was last referenced 9 ticks ago and the other was last referenced 1000 ticks ago. We have no way of seeing that. In practice, however, 8 bits is generally enough if a clock tick is around 20 msec. If a page has not been referenced in 160 msec, it probably is not that important.

4.4.8 The Working Set Page Replacement Algorithm

In the purest form of paging, processes are started up with none of their pages in memory. As soon as the CPU tries to fetch the first instruction, it gets a page fault, causing the operating system to bring in the page containing the first instruction. Other page faults for global variables and the stack usually follow quickly. After a while, the process has most of the pages it needs and settles down to run with relatively few page faults. This strategy is called **demand paging** because pages are loaded only on demand, not in advance.

Of course, it is easy enough to write a test program that systematically reads all the pages in a large address space, causing so many page faults that there is not enough memory to hold them all. Fortunately, most processes do not work this way. They exhibit a **locality of reference**, meaning that during any phase of execution, the process references only a relatively small fraction of its pages. Each pass of a multipass compiler, for example, references only a fraction of all the pages, and a different fraction at that.

The set of pages that a process is currently using is called its **working set** (Denning, 1968a; Denning, 1980). If the entire working set is in memory, the process will run without causing many faults until it moves into another execution phase (e.g., the next pass of the compiler). If the available memory is too small to hold the entire working set, the process will cause many page faults and run slowly since executing an instruction takes a few nanoseconds and reading in a page from the disk typically takes 10 milliseconds. At a rate of one or two instructions per 10 milliseconds, it will take ages to finish. A program causing page faults every few instructions is said to be **thrashing** (Denning, 1968b).

In a multiprogramming system, processes are frequently moved to disk (i.e., all their pages are removed from memory) to let other processes have a turn at the CPU. The question arises of what to do when a process is brought back in again. Technically, nothing need be done. The process will just cause page faults until its working set has been loaded. The problem is that having 20, 100, or even 1000 page faults every time a process is loaded is slow, and it also wastes considerable CPU time, since it takes the operating system a few milliseconds of CPU time to process a page fault.

Therefore, many paging systems try to keep track of each process' working set and make sure that it is in memory before letting the process run. This approach is called the **working set model** (Denning, 1970). It is designed to greatly reduce the page fault rate. Loading the pages *before* letting processes run is also called **prepaging.** Note that the working set changes over time.

It has been long known that most programs do not reference their address space uniformly, but that the references tend to cluster on a small number of pages. A memory reference may fetch an instruction, it may fetch data, or it may store data. At any instant of time, t, there exists a set consisting of all the pages used by the k most recent memory references. This set, $w(k, t)$, is the working set. Because the $k = 1$ most recent references must have used all the pages used by the $k > 1$ most recent references, and possibly others, $w(k, t)$ is a monotonically nondecreasing function of k. The limit of $w(k, t)$ as k becomes large is finite because a program cannot reference more pages than its address space contains, and few programs will use every single page. Figure 4-20 depicts the size of the working set as a function of k.

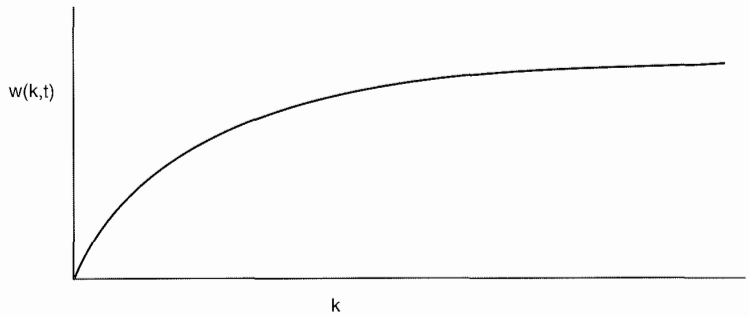


Figure 4-20. The working set is the set of pages used by the k most recent memory references. The function $w(k, t)$ is the size of the working set at time t.

The fact that most programs randomly access a small number of pages, but that this set changes slowly in time explains the initial rapid rise of the curve and then the slow rise for large k. For example, a program that is executing a loop occupying two pages using data on four pages, may reference all six pages every 1000 instructions, but the most recent reference to some other page may be a million instructions earlier, during the initialization phase. Due to this asymptotic behavior, the contents of the working set is not sensitive to the value of k chosen. To put it differently, there exists a wide range of k values for which the working set is unchanged. Because the working set varies slowly with time, it is possible to make a reasonable guess as to which pages will be needed when the program is restarted on the basis of its working set when it was last stopped. Prepaging consists of loading these pages before the process is allowed to run again.

To implement the working set model, it is necessary for the operating system to keep track of which pages are in the working set. Having this information also immediately leads to a possible page replacement algorithm: when a page fault occurs, find a page not in the working set and evict it. To implement such an algorithm, we need a precise way of determining which pages are in the working set and which are not at any given moment in time.

As we mentioned above, the working set is the set of pages used in the k most recent memory references (some authors use the k most recent page references, but the choice is arbitrary). To implement any working set algorithm, some value of k must be chosen in advance. Once some value has been selected, after every memory reference, the set of pages used by the previous k memory references is uniquely determined.

Of course, having an operational definition of the working set does not mean that there is an efficient way to monitor it in real time, during program execution. One could imagine a shift register of length k, with every memory reference shifting the register left one position and inserting the most recently referenced page number on the right. The set of all k page numbers in the shift register would be the working set. In theory, at a page fault, the contents of the shift register could be read out and sorted. Duplicate pages could then be removed. The result would be the working set. However, maintaining the shift register and processing it at a page fault would both be prohibitively expensive, so this technique is never used.

Instead, various approximations are used. One commonly used approximation is to drop the idea of counting back k memory references and use execution time instead. For example, instead of defining the working set as those pages used during the previous 10 million memory references, we can define it as the set of pages used during the past 100 msec of execution time. In practice, such a definition is just as good and much easier to use. Note that for each process, only its own execution time counts. Thus if a process starts running at time T and has had 40 msec of CPU time at real time $T + 100$ msec, for working set purposes, its time is 40 msec. The amount of CPU time a process has actually used has since it started is often called its **current virtual time**. With this approximation, the working set of a process is the set of pages it has referenced during the past τ seconds of virtual time.

Now let us look at a page replacement algorithm based on the working set. The basic idea is to find a page that is not in the working set and evict it. In Fig. 4-21 we see a portion of a page table for some machine. Because only pages that are in memory are considered as candidates for eviction, pages that are absent from memory are ignored by this algorithm. Each entry contains (at least) two items of information: the approximate time the page was last used and the R (Referenced) bit. The empty white rectangle symbolizes the other fields not needed for this algorithm, such as the page frame number, the protection bits, and the M (Modified) bit.

The algorithm works as follows. The hardware is assumed to set the R and M bits, as we have discussed before. Similarly, a periodic clock interrupt is assumed to cause software to run that clears the *Referenced* bit on every clock tick. On every page fault, the page table is scanned to look for a suitable page to evict.

As each entry is processed, the R bit is examined. If it is 1, the current virtual time is written into the *Time of last use* field in the page table, indicating that the page was in use at the time the fault occurred. Since the page has been referenced

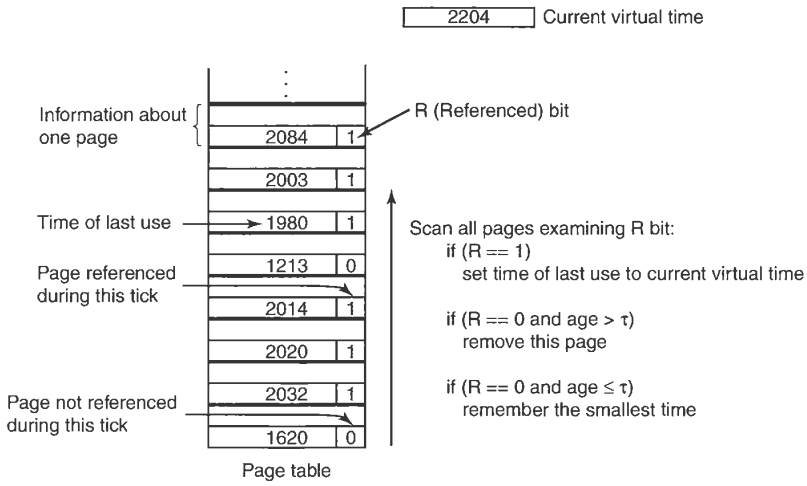


Figure 4-21. The working set algorithm.

during the current clock tick, it is clearly in the working set and is not a candidate for removal (τ is assumed to span multiple clock ticks).

If R is 0, the page has not been referenced during the current clock tick and may be a candidate for removal. To see whether or not it should be removed, its age, that is, the current virtual time minus its *Time of last use* is computed and compared to τ. If the age is greater than τ, the page is no longer in the working set. It is reclaimed and the new page loaded here. The scan continues updating the remaining entries, however.

However, if R is 0 but the age is less than or equal to τ, the page is still in the working set. The page is temporarily spared, but the page with the greatest age (smallest value of *Time of last use*) is noted. If the entire table is scanned without finding a candidate to evict, that means that all pages are in the working set. In that case, if one or more pages with $R = 0$ were found, the one with the greatest age is evicted. In the worst case, all pages have been referenced during the current clock tick (and thus all have $R = 1$), so one is chosen at random for removal, preferably a clean page, if one exists.

4.4.9 The WSClock Page Replacement Algorithm

The basic working set algorithm is cumbersome since the entire page table has to be scanned at each page fault until a suitable candidate is located. An improved algorithm, that is based on the clock algorithm but also uses the working set information is called **WSClock** (Carr and Hennessey, 1981). Due to its simplicity of implementation and good performance, it is widely used in practice.

The data structure needed is a circular list of page frames, as in the clock algorithm, and as shown in Fig. 4-22(a). Initially, this list is empty. When the first page is loaded, it is added to the list. As more pages are added, they go into the list to form a ring. Each entry contains the *Time of last use* field from the basic working set algorithm, as well as the *R* bit (shown) and the *M* bit (not shown).

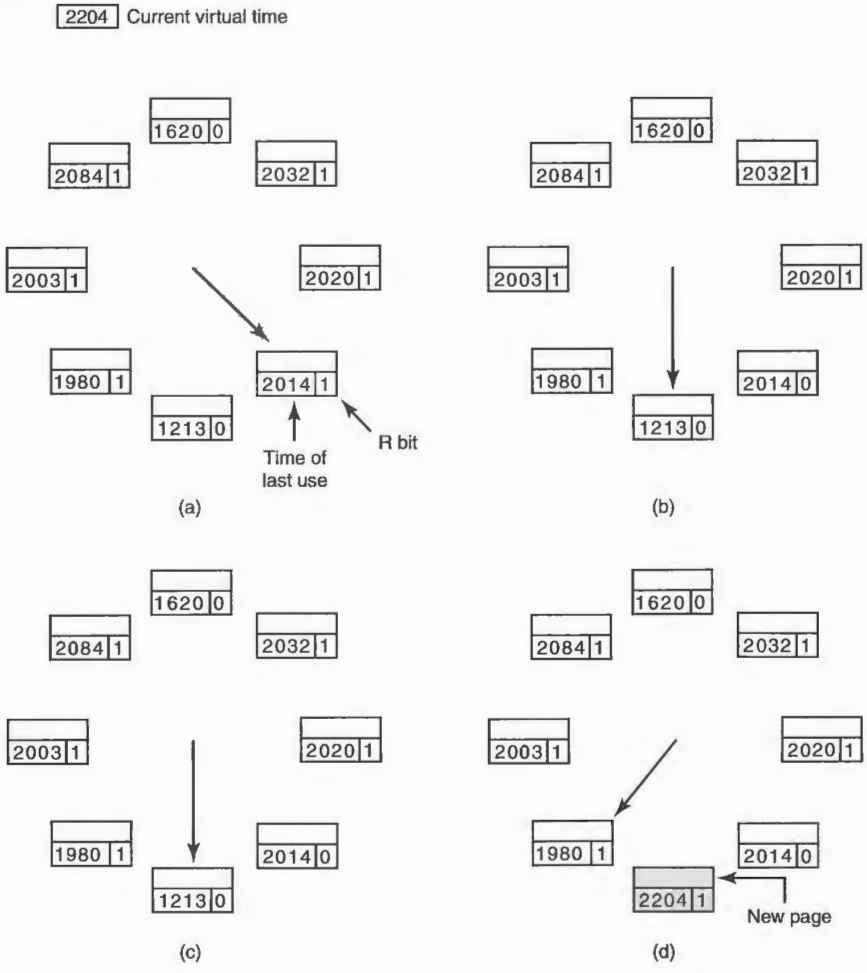


Figure 4-22. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when $R = 1$. (c) and (d) give an example of $R = 0$.

As with the clock algorithm, at each page fault the page pointed to by the hand is examined first. If the *R* bit is set to 1, the page has been used during the

current tick so it is not an ideal candidate to remove. The R bit is then set to 0, the hand advanced to the next page, and the algorithm repeated for that page. The state after this sequence of events is shown in Fig. 4-22(b).

Now consider what happens if the page pointed to has $R = 0$, as shown in Fig. 4-22(c). If the age is greater than τ and the page is clean, it is not in the working set and a valid copy exists on the disk. The page frame is simply claimed and the new page put there, as shown in Fig. 4-22(d). On the other hand, if the page is dirty, it cannot be claimed immediately since no valid copy is present on disk. To avoid a process switch, the write to disk is scheduled, but the hand is advanced and the algorithm continues with the next page. After all, there might be an old, clean page further down the line that can be used immediately.

In principle, all pages might be scheduled for disk I/O on one cycle around the clock. To reduce disk traffic, a limit might be set, allowing a maximum of n pages to be written back. Once this limit has been reached, no new writes are scheduled.

What happens if the hand comes all the way around to its starting point? There are two cases to distinguish:

1. At least one write has been scheduled.
2. No writes have been scheduled.

In the former case, the hand just keeps moving, looking for a clean page. Since one or more writes have been scheduled, eventually some write will complete and its page will be marked as clean. The first clean page encountered is evicted. This page is not necessarily the first write scheduled because the disk driver may reorder writes in order to optimize disk performance.

In the latter case, all pages are in the working set, otherwise at least one write would have been scheduled. Lacking additional information, the simplest thing to do is claim any clean page and use it. The location of a clean page could be kept track of during the sweep. If no clean pages exist, then the current page is chosen and written back to disk.

4.4.10 Summary of Page Replacement Algorithms

We have now looked at a variety of page replacement algorithms. In this section we will briefly summarize them. The list of algorithms discussed is given in Fig. 4-23.

The optimal algorithm replaces the page referenced last among the current pages. Unfortunately, there is no way to determine which page will be last, so in practice this algorithm cannot be used. It is useful as a benchmark against which other algorithms can be measured, however.

The NRU algorithm divides pages into four classes depending on the state of the R and M bits. A random page from the lowest numbered class is chosen. This algorithm is easy to implement, but it is very crude. Better ones exist.

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Figure 4-23. Page replacement algorithms discussed in the text.

FIFO keeps track of the order pages were loaded into memory by keeping them in a linked list. Removing the oldest page then becomes trivial, but that page might still be in use, so FIFO is a bad choice.

Second chance is a modification to FIFO that checks if a page is in use before removing it. If it is, the page is spared. This modification greatly improves the performance. Clock is simply a different implementation of second chance. It has the same performance properties, but takes a little less time to execute the algorithm.

LRU is an excellent algorithm, but it cannot be implemented without special hardware. If this hardware is not available, it cannot be used. NFU is a crude attempt to approximate LRU. It is not very good. However, aging is a much better approximation to LRU and can be implemented efficiently. It is a good choice.

The last two algorithms use the working set. The working set algorithm is reasonable performance, but it is somewhat expensive to implement. WSClock is a variant that not only gives good performance but is also efficient to implement.

All in all, the two best algorithms are aging and WSClock. They are based on LRU and the working set, respectively. Both give good paging performance and can be implemented efficiently. A few other algorithms exist, but these two are probably the most important in practice.

4.5 MODELING PAGE REPLACEMENT ALGORITHMS

Over the years, some work has been done on modeling page replacement algorithms from a theoretical perspective. In this section we will discuss some of these ideas, just to see how the modeling process works.

4.5.1 Belady's Anomaly

Intuitively, it might seem that the more page frames the memory has, the fewer page faults a program will get. Surprisingly enough, this is not always the case. Belady et al. (1969) discovered a counterexample, in which FIFO caused more page faults with four page frames than with three. This strange situation has become known as **Belady's anomaly**. It is illustrated in Fig. 4-24 for a program with five virtual pages, numbered from 0 to 4. The pages are referenced in the order

0 1 2 3 0 1 4 0 1 2 3 4

In Fig. 4-24(a) we see how with three page frames a total of nine page faults are caused. In Fig. 4-24(b) we get ten page faults with four page frames.

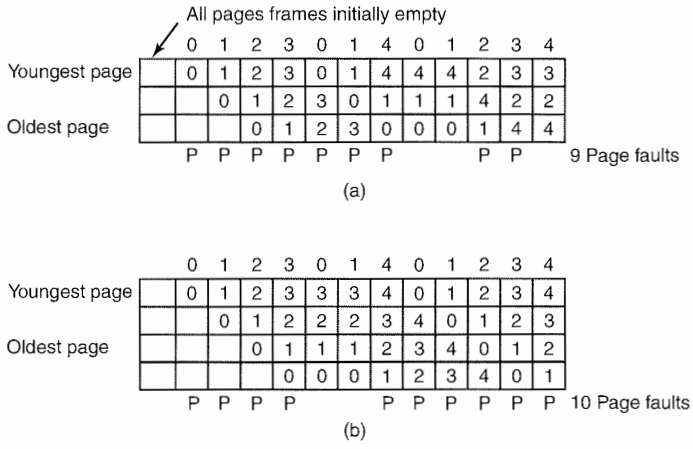


Figure 4-24. Belady's anomaly. (a) FIFO with three page frames. (b) FIFO with four page frames. The *P*'s show which page references cause page faults.

4.5.2 Stack Algorithms

Many researchers in computer science were dumbfounded by Belady's anomaly and began investigating it. This work led to the development of a whole theory of paging algorithms and their properties. While most of this work is beyond the scope of this book, we will give a brief introduction below. For more details see (Maekawa et al., 1987).

All of this work begins with the observation that every process generates a sequence of memory references as it runs. Each memory reference corresponds to

a specific virtual page. Thus conceptually, a process' memory access can be characterized by an (ordered) list of page numbers. This list is called the **reference string**, and plays a central role in the theory. For simplicity, in the rest of this section we will consider only the case of a machine with one process, so each machine has a single, deterministic reference string (with multiple processes, we would have to take into account the interleaving of their reference strings due to the multiprogramming).

A paging system can be characterized by three items:

1. The reference string of the executing process.
2. The page replacement algorithm.
3. The number of page frames available in memory, m.

Conceptually, we can imagine an abstract interpreter that works as follows. It maintains an internal array, M, that keeps track of the state of memory. It has as many elements as the process has virtual pages, which we will call n. The array M is divided into two parts. The top part, with m entries, contains all the pages that are currently in memory. The bottom part, with $n - m$ pages, contains all the pages that have been referenced once but have been paged out and are not currently in memory. Initially, M is the empty set, since no pages have been referenced and no pages are in memory.

As execution begins, the process begins emitting the pages in the reference string, one at a time. As each one comes out, the interpreter checks to see if the page is in memory (i.e., in the top part of M). If it is not, a page fault occurs. If there is an empty slot in memory (i.e., the top part of M contains fewer than m entries), the page is loaded and entered in the top part of M. This situation arises only at the start of execution. If memory is full (i.e., the top part of M contains m entries), the page replacement algorithm is invoked to remove a page from memory. In the model, what happens is that one page is moved from the top part of M to the bottom part, and the needed page entered into the top part. In addition, the top part and the bottom part may be separately rearranged.

To make the operation of the interpreter clearer, let us look at a concrete example using LRU page replacement. The virtual address space has eight pages and the physical memory has four page frames. At the top of Fig. 4-25 we have a reference string consisting of the 24 pages:

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 4 1

Under the reference string, we have 25 columns of 8 items each. The first column, which is empty, reflects the state of M before execution begins. Each successive column shows M after one page has been emitted by the reference and processed by the paging algorithm. The heavy outline denotes the top of M, that is, the first four slots, which correspond to page frames in memory. Pages inside the heavy box are in memory, and pages below it have been paged out to disk.

Reference string	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Page faults	P	P	P	P	P	P	P		P					P			P						P	
Distance string	∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

Figure 4-25. The state of the memory array, M, after each item in the reference string is processed. The distance string will be discussed in the next section.

The first page in the reference string is 0, so it is entered in the top of memory, as shown in the second column. The second page is 2, so it is entered at the top of the third column. This action causes 0 to move down. In this example, a newly loaded page is always entered at the top, and everything else moved down, as needed.

Each of the first seven pages in the reference string causes a page fault. The first four can be handled without removing a page, but starting with the reference to page 5, loading a new page requires removing an old page.

The second reference to page 3 does not cause a page fault, because 3 is already in memory. Nevertheless, the interpreter removes it from where it was and puts it on the top, as shown. The process continues for a while, until page 5 is referenced. This page is moved from bottom part of M to the top part (i.e., it is loaded into memory from disk). Whenever a page is referenced that is not within the heavy box, a page fault occurs, as indicated by the P's below the matrix.

Let us now briefly summarize some of the properties of this model. First, when a page is referenced, it is always moved to the top entry in M. Second, if the page referenced was already in M, all pages above it move down one position. A transition from within the box to outside of it corresponds to a page being evicted from memory. Third, pages that were below the referenced page are not moved. In this manner, the contents of M exactly represent the contents of the LRU algorithm.

Although this example uses LRU, the model works equally well with other algorithms. In particular, there is one class of algorithms that is especially interesting: algorithms that have the property

$$M(m, r) \subseteq M(m + 1, r)$$

where m varies over the page frames and r is an index into the reference string. What this says is that the set of pages included in the top part of M for a memory

with m page frames after r memory references are also included in M for a memory with $m + 1$ page frames. In other words, if we increase memory size by one page frame and re-execute the process, at every point during the execution, all the pages that were present in the first run are also present in the second run, along with one additional page.

From examination of Fig. 4-25 and a little thought about how it works, it should be clear that LRU has this property. Some other algorithms (e.g., optimal page replacement) also have it, but FIFO does not. Algorithms that have this property are called **stack algorithms**. These algorithms do not suffer from Belady's anomaly and are thus much loved by virtual memory theorists.

4.5.3 The Distance String

For stack algorithms, it is often convenient to represent the reference string in a more abstract way than the actual page numbers. A page reference will be henceforth denoted by the distance from the top of the stack where the referenced page was located. For example, the reference to page 1 in the last column of Fig. 4-25 is a reference to a page at a distance 3 from the top of the stack (because page 1 was in third place *before* the reference). Pages that have not yet been referenced and thus are not yet on the stack (i.e., not yet in M) are said to be at a distance ∞. The distance string for Fig. 4-25 is given at the bottom of the figure.

Note that the distance string depends not only on the reference string, but also on the paging algorithm. With the same original reference string, a different paging algorithm would make different choices about which pages to evict. As a result, a different sequence of stacks arises.

The statistical properties of the distance string have a big impact on the performance of the algorithm. In Fig. 4-26(a) we see the probability density function for the entries in a (ficticious) distance string, d. Most of the entries in the string are between 1 and k. With a memory of k page frames, few page faults occur.

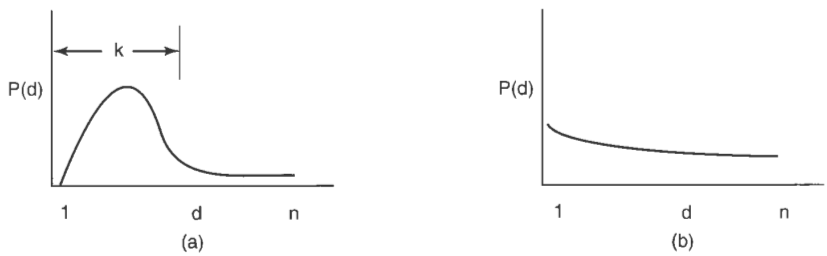


Figure 4-26. Probability density functions for two hypothetical distance strings.

In contrast, in Fig. 4-26(b), the references are so spread out that the only way to avoid a large number of page faults is to give the program as many page frames as it has virtual pages. Having a program like this is just bad luck.

4.5.4 Predicting Page Fault Rates

One of the nice properties of the distance string is that it can be used to predict the number of page faults that will occur with memories of different sizes. We will demonstrate how this computation can be made based on the example of Fig. 4-25. The goal is to make one pass over the distance string and, from the information collected, to be able to predict how many page faults the process would have in memories with 1, 2, 3, ..., n page frames, where n is the number of virtual pages in the process' address space.

The algorithm starts by scanning the distance string, page by page. It keeps track of the number of times 1 occurs, the number of times 2 occurs, and so on. Let C_i be the number of occurrences of i. For the distance string of Fig. 4-25, the C vector is illustrated in Fig. 4-27(a). In this example, it happens four times that the page referenced is already on top of the stack. Three times the reference is to the next-to-the-top page, and so forth. Let C_∞ be the number of times ∞ occurs in the distance string.

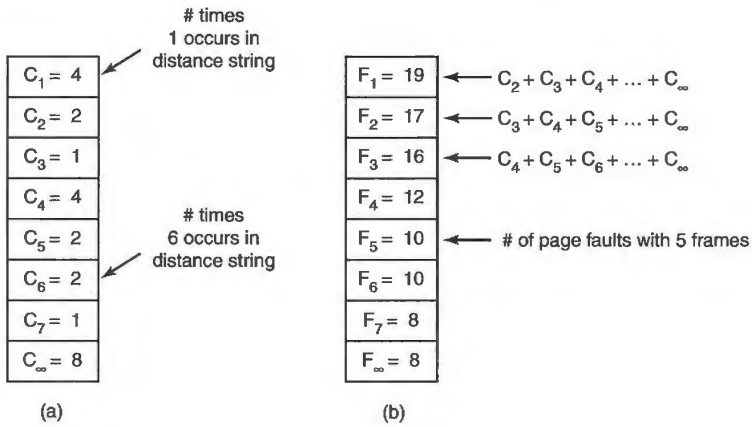


Figure 4-27. Computation of the page fault rate from the distance string. (a) The C vector. (b) F vector.

Now compute the F vector according to the formula

$$F_m = \sum_{k=m+1}^{n} C_k + C_\infty$$

The value of F_m is the number of page faults that will occur with the given distance string and m page frames. For the distance string of Fig. 4-25, Fig. 4-27(b) gives the F vector. For example, F_1 is 20, meaning that with a memory holding only 1 page frame, out of the 24 references in the string, all get page faults except the four that are the same as the previous page reference.

To see why this formula works, go back to the heavy box in Fig. 4-25. Let m be the number of page frames in the top part of M. A page fault occurs any time an element of the distance string is $m + 1$ or more. The summation in the formula above adds up the number of times such elements occur. This model can be used to make other predictions as well (Maekawa et al., 1987).

4.6 DESIGN ISSUES FOR PAGING SYSTEMS

In the previous sections we have explained how paging works and have given a few of the basic page replacement algorithms and shown how to model them. But knowing the bare mechanics is not enough. To design a system, you have to know a lot more to make it work well. It is like the difference between knowing how to move the rook, knight, bishop, and other pieces in chess, and being a good player. In the following sections, we will look at other issues that operating system designers must consider carefully in order to get good performance from a paging system.

4.6.1 Local versus Global Allocation Policies

In the preceding sections we have discussed several algorithms for choosing a page to replace when a fault occurs. A major issue associated with this choice (which we have carefully swept under the rug until now) is how memory should be allocated among the competing runnable processes.

Take a look at Fig. 4-28(a). In this figure, three processes, *A*, *B*, and *C*, make up the set of runnable processes. Suppose *A* gets a page fault. Should the page replacement algorithm try to find the least recently used page considering only the six pages currently allocated to *A*, or should it consider all the pages in memory? If it looks only at *A*'s pages, the page with the lowest age value is *A5*, so we get the situation of Fig. 4-28(b).

On the other hand, if the page with the lowest age value is removed without regard to whose page it is, page *B3* will be chosen and we will get the situation of Fig. 4-28(c). The algorithm of Fig. 4-28(b) is said to be a **local** page replacement algorithm, whereas that of Fig. 4-28(c) is said to be a **global** algorithm. Local algorithms effectively correspond to allocating every process a fixed fraction of the memory. Global algorithms dynamically allocate page frames among the runnable processes. Thus the number of page frames assigned to each process varies in time.

In general, global algorithms work better, especially when the working set size can vary over the lifetime of a process. If a local algorithm is used and the working set grows, thrashing will result, even if there are plenty of free page frames. If the working set shrinks, local algorithms waste memory. If a global algorithm is used, the system must continually decide how many page frames to assign to each process. One way is to monitor the working set size as indicated by

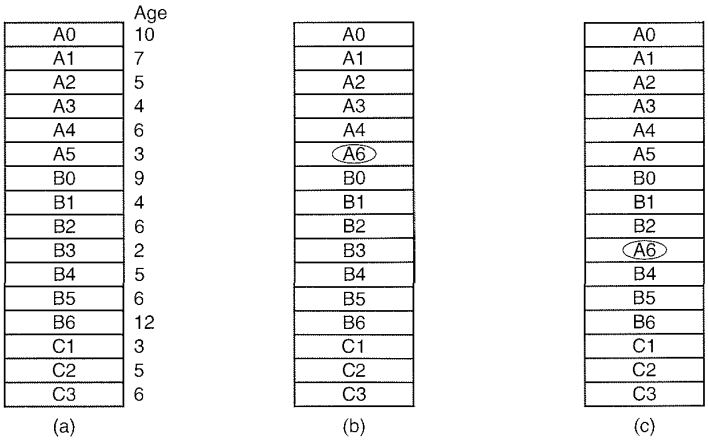


Figure 4-28. Local versus global page replacement. (a) Original configuration. (b) Local page replacement. (c) Global page replacement.

the aging bits, but this approach does not necessarily prevent thrashing. The working set may change size in microseconds, whereas the aging bits are a crude measure spread over a number of clock ticks.

Another approach is to have an algorithm for allocating page frames to processes. One way is to periodically determine the number of running processes and allocate each process an equal share. Thus with 12,416 available (i.e., non-operating system) page frames and 10 processes, each process gets 1241 frames. The remaining 6 go into a pool to be used when page faults occur.

Although this method seems fair, it makes little sense to give equal shares of the memory to a 10-KB process and a 300-KB process. Instead, pages can be allocated in proportion to each process' total size, with a 300-KB process getting 30 times the allotment of a 10-KB process. It is probably wise to give each process some minimum number, so it can run, no matter how small it is. On some machines, for example, a single two-operand instruction may need as many as six pages because the instruction itself, the source operand, and the destination operand may all straddle page boundaries. With an allocation of only five pages, programs containing such instructions cannot execute at all.

If a global algorithm is used, it may be possible to start each process up with some number of pages proportional to the process' size, but the allocation has to be updated dynamically as the processes run. One way to manage the allocation is to use the **PFF** (**Page Fault Frequency**) algorithm. It tells when to increase or decrease a process' page allocation but says nothing about which page to replace on a fault. It just controls the size of the allocation set.

For a large class of page replacement algorithms, including LRU, it is known that the fault rate decreases as more pages are assigned, as we discussed above. This is the assumption behind PFF. This property is illustrated in Fig. 4-29.

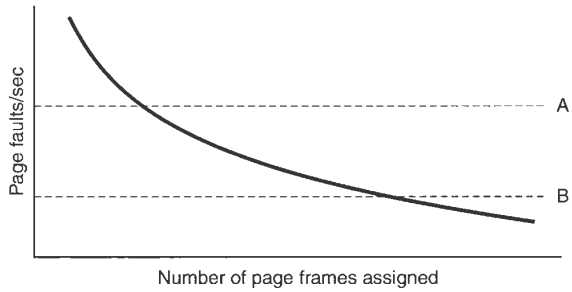


Figure 4-29. Page fault rate as a function of the number of page frames assigned.

Measuring the page fault rate is straightforward: just count the number of faults per second, possibly taking a running mean over past seconds as well. One easy way to do this is to add the present second's value to the current running mean and divide by two. The dashed line marked *A* corresponds to a page fault rate that is unacceptably high, so the faulting process is given more page frames to reduce the fault rate. The dashed line marked *B* corresponds to a page fault rate so low that it can be concluded that the process has too much memory. In this case page frames may be taken away from it. Thus, PFF tries to keep the paging rate for each process within acceptable bounds.

It is important to note that some page replacement algorithms can work with either a local replacement policy or a global one. For example, FIFO can replace the oldest page in all of memory (global algorithm) or the oldest page owned by the current process (local algorithm). Similarly, LRU or some approximation to it can replace the least recently page in all of memory (global algorithm) or the least recently used page owned by the current process (local algorithm). The choice of local versus global is independent of the algorithm in some cases.

On the other hand, for other page replacement algorithms, only a local strategy makes sense. In particular, the working set and WSClock algorithms refer to some specific process and must be applied in that context. There really is no working set for the machine as a whole and trying to use the union of all the working sets would lose the locality property and not work well.

4.6.2 Load Control

Even with the best page replacement algorithm and optimal global allocation of page frames to processes, it can happen that the system thrashes. In fact, whenever the combined working sets of all processes exceed the capacity of memory,

thrashing can be expected. One symptom of this situation is that the PFF algorithm indicates that some processes need more memory but no processes need less memory. In this case there is no way to give more memory to those processes needing it without hurting some other processes. The only real solution is to temporarily get rid of some processes.

The way to reduce the number of processes competing for memory is to swap some of them to the disk and free up all the pages they are holding. For example, one process can be swapped to disk and its page frames divided up among other processes that are thrashing. If the thrashing stops, the system can run for a while this way. If it does not stop, another process has to be swapped out, and so on, until the thrashing stops. Thus even with paging, swapping is still needed, only now swapping is used to reduce potential demand for memory, rather than to reclaim blocks of it for immediate use.

Swapping processes out to relieve the load on memory is reminiscent of two-level scheduling, in which some processes are put on disk and a short-term scheduler is used to schedule the remaining processes. Clearly, the two ideas can be combined, with just enough processes swapped out to make the page-fault rate acceptable. Periodically, some processes are brought in from disk and other ones are swapped out there.

However, another factor to consider is the degree of multiprogramming. As we saw in Fig. 4-4, when the number of processes in main memory is too low, the CPU may be idle for substantial periods of time. This consideration argues for considering not only process size and paging rate when deciding which process to swap out, but also its characteristics, such as whether it is CPU bound or I/O bound, and what characteristics the remaining processes have as well.

4.6.3 Page Size

The page size is often a parameter that can be chosen by the operating system. Even if the hardware has been designed with, for example, 512-byte pages, the operating system can easily regard pages 0 and 1, 2 and 3, 4 and 5, and so on, as 1-KB pages by always allocating two consecutive 512-byte page frames for them.

Determining the best page size requires balancing several competing factors. As a result, there is no overall optimum. To start with, there are two factors that argue for a small page size. A randomly chosen text, data, or stack segment will not fill an integral number of pages. On the average, half of the final page will be empty. The extra space in that page is wasted. This wastage is called **internal fragmentation**. With n segments in memory and a page size of p bytes, $np/2$ bytes will be wasted on internal fragmentation. This reasoning argues for a small page size.

Another argument for a small page size becomes apparent if we think about a program consisting of eight sequential phases of 4 KB each. With a 32-KB page

size, the program must be allocated 32 KB all the time. With a 16-KB page size, it needs only 16 KB. With a page size of 4 KB or smaller, it requires only 4 KB at any instant. In general, a large page size will cause more unused program to be in memory than a small page size.

On the other hand, small pages mean that programs will need many pages, hence a large page table. A 32-KB program needs only four 8-KB pages, but 64 512-byte pages. Transfers to and from the disk are generally a page at a time, with most of the time being for the seek and rotational delay, so that transferring a small page takes almost as much time as transferring a large page. It might take 64×10 msec to load 64 512-byte pages, but only 4×12 msec to load four 8-KB pages.

On some machines, the page table must be loaded into hardware registers every time the CPU switches from one process to another. On these machines having a small page size means that the time required to load the page registers gets longer as the page size gets smaller. Furthermore, the space occupied by the page table increases as the page size decreases.

This last point can be analyzed mathematically. Let the average process size be s bytes and the page size be p bytes. Furthermore, assume that each page entry requires e bytes. The approximate number of pages needed per process is then s/p, occupying se/p bytes of page table space. The wasted memory in the last page of the process due to internal fragmentation is $p/2$. Thus, the total overhead due to the page table and the internal fragmentation loss is given by the sum of these two terms:

$$\text{overhead} = se/p + p/2$$

The first term (page table size) is large when the page size is small. The second term (internal fragmentation) is large when the page size is large. The optimum must lie somewhere in between. By taking the first derivative with respect to p and equating it to zero, we get the equation

$$-se/p^2 + 1/2 = 0$$

From this equation we can derive a formula that gives the optimum page size (considering only memory wasted in fragmentation and page table size). The result is:

$$p = \sqrt{2se}$$

For s = 1MB and e = 8 bytes per page table entry, the optimum page size is 4 KB. Commercially available computers have used page sizes ranging from 512 bytes to 64 KB. A typical value used to 1 KB, but nowadays 4 KB or 8 KB are more common. As memories get larger, the page size tends to get larger as well (but not linearly). Quadrupling the RAM size rarely even doubles the page size.

4.6.4 Separate Instruction and Data Spaces

Most computers have a single address space that holds both programs and data, as shown in Fig. 4-30(a). If this address space is large enough, everything works fine. However, it is often too small, forcing programmers to stand on their heads to fit everything into the address space.

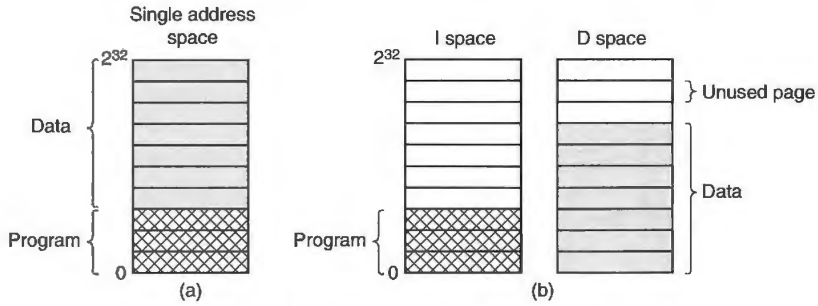


Figure 4-30. (a) One address space. (b) Separate I and D spaces.

One solution, pioneered on the (16-bit) PDP-11, is to have separate address spaces for instructions (program text) and data. These are called **I-space** and **D-space**, respectively. Each address space runs from 0 to some maximum, typically $2^{16} - 1$ or $2^{32} - 1$. In Fig. 4-30(b) we see both of them. The linker must know when separate I- and D-spaces are being used because when they are, the data are relocated to virtual address 0 instead of starting after the program.

In a computer with this design, both address spaces can be paged, independently from one another. Each one has its own page table, with its own mapping of virtual pages to physical page frames. When the hardware wants to fetch an instruction, it knows that it must use I-space and the I-space page table. Similarly, references to data must go through the D-space page table. Other than this distinction, having separate I- and D-spaces does not introduce any special complications and it does double the available address space.

4.6.5 Shared Pages

Another design issue is sharing. In a large multiprogramming system, it is common for several users to be running the same program at the same time. It is clearly more efficient to share the pages, to avoid having two copies of the same page in memory at the same time. One problem is that not all pages are sharable. In particular, pages that are read-only, such as program text, can be shared, but data pages cannot.

If separate I- and D-spaces are supported, it is relatively straightforward to share programs by having two or more processes use the same page table for their

I-space but different page tables for their D-spaces. Typically in an implementation that supports sharing in this way, page tables are data structures independent of the process table. Each process then has two pointers in its process table: one to the I-space page table and one to the D-space page table, as shown in Fig. 4-31. When the scheduler chooses a process to run, it uses these pointers to locate the appropriate page tables and sets up the MMU using them. Even without separate I- and D-spaces, processes can share programs (or sometimes, libraries), but the mechanism is more complicated.

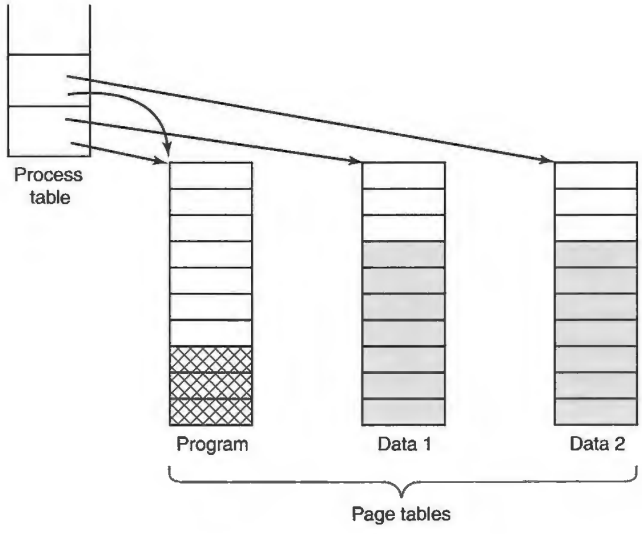


Figure 4-31. Two processes sharing the same program sharing its page table.

When two or more processes share some code, a problem occurs with the shared pages. Suppose that processes *A* and *B* are both running the editor and sharing its pages. If the scheduler decides to remove *A* from memory, evicting all its pages and filling the empty page frames with some other program will cause *B* to generate a large number of page faults to bring them back in again.

Similarly, when *A* terminates, it is essential to be able to discover that the pages are still in use so that their disk space will not be freed by accident. Searching all the page tables to see if a page is shared is usually too expensive, so special data structures are needed to keep track of shared pages, especially if the unit of sharing is the individual page (or run of pages), rather than an entire page table.

Sharing data is trickier than sharing code, but it is not impossible. In particular, in UNIX, after a fork system call, the parent and child are required to share both program text and data. In a paged system, what is often done is to give each of these processes its own page table and have both of them point to the same set

of pages. Thus no copying of pages is done at fork time. However, all the data pages are mapped into both processes as READ ONLY.

As long as both processes just read their data, without modifying it, this situation can continue. As soon as either process updates a memory word, the violation of the read-only protection causes a trap to the operating system. A copy is then made of the page so that each process now has its own private copy. Both copies are now set to READ-WRITE so subsequent writes to either copy proceed without trapping. This strategy means that those pages that are never written (including all the program pages) need not be copied. Only the data pages that are actually written need be copied. This approach, called **copy on write**, improves performance by reducing copying.

4.6.6 Cleaning Policy

Paging works best when there are plenty of free page frames that can be claimed as page faults occur. If every page frame is full, and furthermore modified, before a new page can be brought in, an old page must first be written to disk. To insure a plentiful supply of free page frames, many paging systems have a background process, called the **paging daemon**, that sleeps most of the time but is awakened periodically to inspect the state of memory. If too few page frames are free, the paging daemon begins selecting pages to evict using the chosen page replacement algorithm. If these pages have been modified since being loaded, they are written to disk.

In any event, the previous contents of the page are remembered. In the event one of the evicted pages is needed again before its frame has been overwritten, it can be reclaimed by removing it from the pool of free page frames. Keeping a supply of page frames around yields better performance than using all of memory and then trying to find a frame at the moment it is needed. At the very least, the paging daemon ensures that all the free frames are clean, so they need not be written to disk in a big hurry when they are required.

One way to implement this cleaning policy is with a two-handed clock. The front hand is controlled by the paging daemon. When it points to a dirty page, that page it written back to disk and the front hand advanced. When it points to a clean page, it is just advanced. The back hand is used for page replacement, as in the standard clock algorithm. Only now, the probability of the back hand hitting a clean page is increased due to the work of the paging daemon.

4.6.7 Virtual Memory Interface

Up until now, our whole discussion has assumed that virtual memory is transparent to processes and programmers, that is, all they see is a large virtual address space on a computer with a small(er) physical memory. With many systems, that is true, but in some advanced systems, programmers have some control over the

memory map and can use it in nontraditional ways to enhance program behavior. In this section, we will briefly look at a few of these.

One reason for giving programmers control over their memory map is to allow two or more processes to share the same memory. If programmers can name regions of their memory, it may be possible for one process to give another process the name of a memory region so that process can also map it in. With two (or more) processes sharing the same pages, high bandwidth sharing becomes possible—one process writes into the shared memory and another one reads from it.

Sharing of pages can also be used to implement a high-performance message-passing system. Normally, when messages are passed, the data are copied from one address space to another, at considerable cost. If processes can control their page map, a message can be passed by having the sending process unmap the page(s) containing the message, and the receiving process mapping them in. Here only the page names have to be copied, instead of all the data.

Yet another advanced memory management technique is **distributed shared memory** (Feeley et al., 1995; Li, 1986; Li and Hudak, 1989; and Zekauskas et al., 1994). The idea here is to allow multiple processes over a network to share a set of pages, possibly, but not necessarily, as a single shared linear address space. When a process references a page that is not currently mapped in, it gets a page fault. The page fault handler, which may be in the kernel or in user space, then locates the machine holding the page and sends it a message asking it to unmap the page and send it over the network. When the page arrives, it is mapped in and the faulting instruction is restarted. We will examine distributed shared memory in more detail in Chap. 8.

4.7 IMPLEMENTATION ISSUES

Implementers of virtual memory systems have to make choices among the major theoretical algorithms such as second chance versus aging, local versus global page allocation, and demand paging versus prepaging. But they also have to be aware of a number of practical implementation issues as well. In this section we will take a look at a few of the more common problems and some solutions.

4.7.1 Operating System Involvement with Paging

There are four times when the operating system has work to do relating to paging: process creation time, process execution time, page fault time, and process termination time. We will now briefly examine each of these to see what has to be done.

When a new process is created in a paging system, the operating system has to determine how large the program and data will be (initially) and create a page

table for it. Space has to be allocated in memory for the page table and it has to be initialized. The page table need not be resident when the process is swapped out but has to be in memory when the process is running. In addition, space has to be allocated in the swap area on disk so that when a page is swapped out, it has somewhere to go. The swap area also has to be initialized with program text and data so that when the new process starts getting page faults, the pages can be brought in from disk. Some systems page the program text directly from the executable file, thus saving disk space and initialization time. Finally, information about the page table and swap area on disk must be recorded in the process table.

When a process is scheduled for execution, the MMU has to be reset for the new process and the TLB flushed, to get rid of traces of the previously executing process. The new process' page table has to be made current, usually by copying it or a pointer to it to some hardware register(s). Optionally, some or all of the process' pages can be brought into memory to reduce the number of page faults initially.

When a page fault occurs, the operating system has to read out hardware registers to determine which virtual address caused the fault. From this information, it must compute which page is needed and locate that page on disk. It must then find an available page frame to put the new page, evicting some old page if need be. Then it must read the needed page into the page frame. Finally, it must back up the program counter to have it point to the faulting instruction and let that instruction execute again.

When a process exits, the operating system must release its page table, its pages, and the disk space that the pages occupy when they are on disk. If some of the pages are shared with other processes, the pages in memory and on disk can only be released when the last process using them has terminated.

4.7.2 Page Fault Handling

We are finally in a position to describe what happens on a page fault in some detail. The sequence of events is as follows:

1. The hardware traps to the kernel, saving the program counter on the stack. On most machines, some information about the state of the current instruction is saved in special CPU registers.

2. An assembly code routine is started to save the general registers and other volatile information, to keep the operating system from destroying it. This routine calls the operating system as a procedure.

3. The operating system discovers that a page fault has occurred, and tries to discover which virtual page is needed. Often one of the hardware registers contains this information. If not, the operating system must retrieve the program counter, fetch the instruction, and parse it in software to figure out what it was doing when the fault hit.

4. Once the virtual address that caused the fault is known, the system checks to see if this address is valid and the protection consistent with the access. If not, the process is sent a signal or killed. If the address is valid and no protection fault has occurred, the system checks to see if a page frame is free. If no frames are free, the page replacement algorithm is run to select a victim.
5. If the page frame selected is dirty, the page is scheduled for transfer to the disk, and a context switch takes place, suspending the faulting process and letting another one run until the disk transfer has completed. In any event, the frame is marked as busy to prevent it from being used for another purpose.
6. As soon as the page frame is clean (either immediately or after it is written to disk), the operating system looks up the disk address where the needed page is, and schedules a disk operation to bring it in. While the page is being loaded, the faulting process is still suspended and another user process is run, if one is available.
7. When the disk interrupt indicates that the page has arrived, the page tables are updated to reflect its position, and the frame is marked as being in normal state.
8. The faulting instruction is backed up to the state it had when it began and the program counter is reset to point to that instruction.
9. The faulting process is scheduled, and the operating system returns to the assembly language routine that called it.
10. This routine reloads the registers and other state information and returns to user space to continue execution, as if no fault had occurred.

4.7.3 Instruction Backup

When a program references a page that is not in memory, the instruction causing the fault is stopped part way through and a trap to the operating system occurs. After the operating system has fetched the page needed, it must restart the instruction causing the trap. This is easier said than done.

To see the nature of this problem at its worst, consider a CPU that has instructions with two addresses, such as the Motorola 680x0, widely used in embedded systems. The instruction

```
MOV.L #6(A1),2(A0)
```

is 6 bytes, for example (see Fig. 4-32). In order to restart the instruction, the operating system must determine where the first byte of the instruction is. The value

of the program counter at the time of the trap depends on which operand faulted and how the CPU's microcode has been implemented.

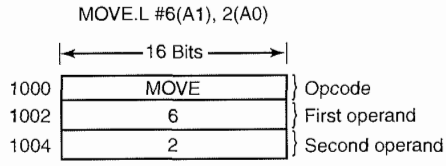


Figure 4-32. An instruction causing a page fault.

In Fig. 4-32, we have an instruction starting at address 1000 that makes three memory references: the instruction word itself, and two offsets for the operands. Depending on which of these three memory references caused the page fault, the program counter might be 1000, 1002, or 1004 at the time of the fault. It is frequently impossible for the operating system to determine unambiguously where the instruction began. If the program counter is 1002 at the time of the fault, the operating system has no way of telling whether the word in 1002 is a memory address associated with an instruction at 1000 (e.g., the location of an operand), or an instruction opcode.

Bad as this problem may be, it could have been worse. Some 680x0 addressing modes use autoincrementing, which means that a side effect of executing the instruction is to increment one or more registers. Instructions that use autoincrement mode can also fault. Depending on the details of the microcode, the increment may be done before the memory reference, in which case the operating system must decrement the register in software before restarting the instruction. Or, the autoincrement may be done after the memory reference, in which case it will not have been done at the time of the trap and must not be undone by the operating system. Autodecrement mode also exists and causes a similar problem. The precise details of whether autoincrements and autodecrements have or have not been done before the corresponding memory references may differ from instruction to instruction and from CPU model to CPU model.

Fortunately, on some machines the CPU designers provide a solution, usually in the form of a hidden internal register into which the program counter is copied just before each instruction is executed. These machines may also have a second register telling which registers have already been autoincremented or autodecremented, and by how much. Given this information, the operating system can unambiguously undo all the effects of the faulting instruction so it can be started all over again. If this information is not available, the operating system has to jump through hoops to figure out what happened and how to repair it. It is as though the hardware designers were unable to solve the problem, so they threw up their hands and told the operating system writers to deal with it. Nice guys.

4.7.4 Locking Pages in Memory

Although we have not discussed I/O much in this chapter, the fact that a computer has virtual memory does not mean that I/O is absent. Virtual memory and I/O interact in subtle ways. Consider a process that has just issued a system call to read from some file or device into a buffer within its address space. While waiting for the I/O to complete, the process is suspended and another process is allowed to run. This other process gets a page fault.

If the paging algorithm is global, there is a small, but nonzero, chance that the page containing the I/O buffer will be chosen to be removed from memory. If an I/O device is currently in the process of doing a DMA transfer to that page, removing it will cause part of the data to be written in the buffer where they belong, and part of the data to be written over the newly loaded page. One solution to this problem is to lock pages engaged in I/O in memory so that they will not be removed. Locking a page is often called **pinning** it in memory. Another solution is to do all I/O to kernel buffers and then copy the data to user pages later.

4.7.5 Backing Store

In our discussion of page replacement algorithms, we saw how a page is selected for removal. We have not said much about where on the disk it is put when it is paged out. Let us now describe some of the issues related to disk management.

The simplest algorithm for allocating page space on the disk is to have a special swap area on the disk. When the system is booted, this area is empty, and is represented in memory as a single entry giving its origin and size. When the first process is started, a chunk of the swap area the size of the first process is reserved and the remaining area reduced by that amount. As new processes are started, they are assigned chunks of the swap area equal in size to their core images. As they finish, their disk space is freed. The swap area is managed as a list of free chunks.

Associated with each process is the disk address of its swap area, kept in the process table. Calculating the address to write a page to becomes simple: just add the offset of the page within the virtual address space to the start of the swap area. However, before a process can start, the swap area must be initialized. One way is to copy the entire process image to the swap area, so that it can be brought *in* as needed. The other is to load the entire process in memory and let it be paged *out* as needed.

However, this simple model has a problem: processes can increase in size after starting. Although the program text is usually fixed, the data area can sometimes grow, and the stack can always grow. Consequently, it may be better to reserve separate swap areas for the text, data, and stack and allow each of these areas to consist of more than one chunk on the disk.

The other extreme is to allocate nothing in advance and allocate disk space for each page when it is swapped out and deallocate it when it is swapped back in. In this way, processes in memory do not tie up any swap space. The disadvantage is that a disk address is needed in memory to keep track of each page on disk. In other words, there must a table per process telling for each page on disk where it is. The two alternatives are shown in Fig. 4-33.

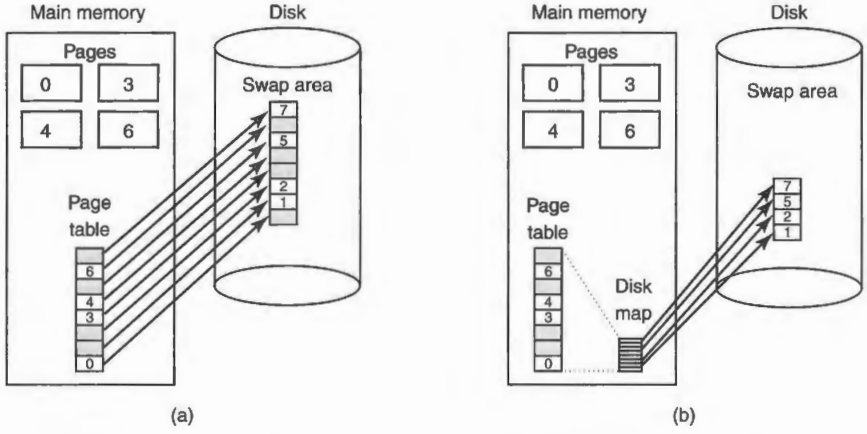


Figure 4-33. (a) Paging to a static swap area. (b) Backing up pages dynamically.

In Fig. 4-33(a), a page table with eight pages is illustrated. Pages 0, 3, 4, and 6 are in main memory. Pages 1, 2, 5, and 7 are on disk. The swap area on disk is as large as the process virtual address space (eight pages) with each page having a fixed location to which it is written when it is evicted from main memory. Calculation of this address requires knowing only where the process' paging area begins, since pages are stored in it contiguously in order of their virtual page number. Pages that are in memory always have a shadow copy on disk, but this copy may be out of date if the page has been modified since being loaded.

In Fig. 4-33(b), pages do not have fixed addresses on disk. When a page is swapped out, an empty disk page is chosen on the fly and the disk map (which has room for one disk address per virtual page) is updated accordingly. Pages that are in memory have no copy on disk. Their entries in the disk map contain an invalid disk address or a bit marking them as not in use.

4.7.6 Separation of Policy and Mechanism

An important tool for managing the complexity of any system is to separate the policy from the mechanism. This principle can be applied to memory management by having most of the memory manager run as a user-level process.

Such a separation was first done in Mach (Young et al., 1987) and in MINIX (Tanenbaum, 1987). The discussion below is loosely based on Mach.

A simple example of how policy and mechanism can be separated is shown in Fig. 4-34. Here the memory management system is divided into three parts:

1. A low-level MMU handler.
2. A page fault handler that is part of the kernel.
3. An external pager running in user space.

All the details of how the MMU works are encapsulated in the MMU handler, which is machine-dependent code and has to be rewritten for each new platform the operating system is ported to. The page-fault handler is machine independent code and contains most of the mechanism for paging. The policy is largely determined by the external pager, which runs as a user process.

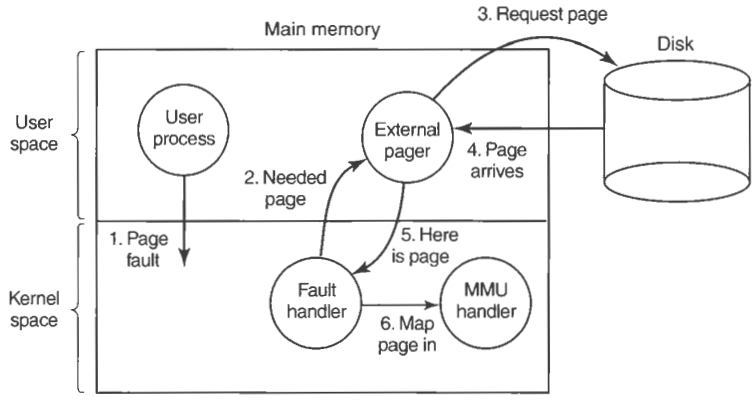


Figure 4-34. Page fault handling with an external pager.

When a process starts up, the external pager is notified in order to set up the process page map and allocate backing store on the disk if need be. As the process runs, it may map new objects into its address space, so the external pager is again notified.

Once the process starts running, it may get a page fault. The fault handler figures out which virtual page is needed and sends a message to the external pager, telling what the problem is. The external pager then reads the needed page in from the disk and copies it to a portion of its own address space. Then it tells the fault handler where the page is. The fault handler then unmaps the page from the external pager's address space and asks the MMU handler to put it into the user's address space at the right place. Then the user process can be restarted.

This implementation leaves open where the page replacement algorithm is put. It would be cleanest to have it in the external pager, but there are some problems with this approach. Principal among these is that the external pager does not have access to the *R* and *M* bits of all the pages. These bits play a role in many of the paging algorithms. Thus either some mechanism is needed to pass this information up to the external pager, or the page replacement algorithm must go in the kernel. In the latter case, the fault handler tells the external pager which page it has selected for eviction and provides the data, either by mapping it into the external pager's address space or including it in a message. Either way, the external pager writes the data to disk.

The main advantage of this implementation is more modular code and greater flexibility. The main disadvantage is the extra overhead of crossing the user-kernel boundary several times and the overhead of the various messages being sent between the pieces of the system. At the moment, the subject is highly controversial, but as computers get faster and faster, and the software gets more and more complex, in the long run sacrificing some performance for more reliable software will probably be acceptable to most implementers.

4.8 SEGMENTATION

The virtual memory discussed so far is one-dimensional because the virtual addresses go from 0 to some maximum address, one address after another. For many problems, having two or more separate virtual address spaces may be much better than having only one. For example, a compiler has many tables that are built up as compilation proceeds, possibly including

1. The source text being saved for the printed listing (on batch systems).
2. The symbol table, containing the names and attributes of variables.
3. The table containing all the integer and floating-point constants used.
4. The parse tree, containing the syntactic analysis of the program.
5. The stack used for procedure calls within the compiler.

Each of the first four tables grows continuously as compilation proceeds. The last one grows and shrinks in unpredictable ways during compilation. In a one-dimensional memory, these five tables would have to be allocated contiguous chunks of virtual address space, as in Fig. 4-35.

Consider what happens if a program has an exceptionally large number of variables but a normal amount of everything else. The chunk of address space allocated for the symbol table may fill up, but there may be lots of room in the other tables. The compiler could, of course, simply issue a message saying that the compilation cannot continue due to too many variables, but doing so does not seem very sporting when unused space is left in the other tables.

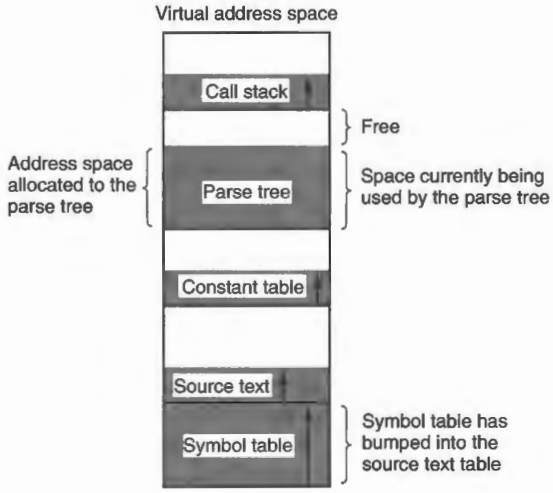


Figure 4-35. In a one-dimensional address space with growing tables, one table may bump into another.

Another possibility is to play Robin Hood, taking space from the tables with an excess of room and giving it to the tables with little room. This shuffling can be done, but it is analogous to managing one's own overlays—a nuisance at best and a great deal of tedious, unrewarding work at worst.

What is really needed is a way of freeing the programmer from having to manage the expanding and contracting tables, in the same way that virtual memory eliminates the worry of organizing the program into overlays.

A straightforward and extremely general solution is to provide the machine with many completely independent address spaces, called **segments**. Each segment consists of a linear sequence of addresses, from 0 to some maximum. The length of each segment may be anything from 0 to the maximum allowed. Different segments may, and usually do, have different lengths. Moreover, segment lengths may change during execution. The length of a stack segment may be increased whenever something is pushed onto the stack and decreased whenever something is popped off the stack.

Because each segment constitutes a separate address space, different segments can grow or shrink independently, without affecting each other. If a stack in a certain segment needs more address space to grow, it can have it, because there is nothing else in its address space to bump into. Of course, a segment can fill up but segments are usually very large, so this occurrence is rare. To specify an address in this segmented or two-dimensional memory, the program must supply a two-part address, a segment number, and an address within the segment.

Figure 4-36 illustrates a segmented memory being used for the compiler tables discussed earlier. Five independent segments are shown here.

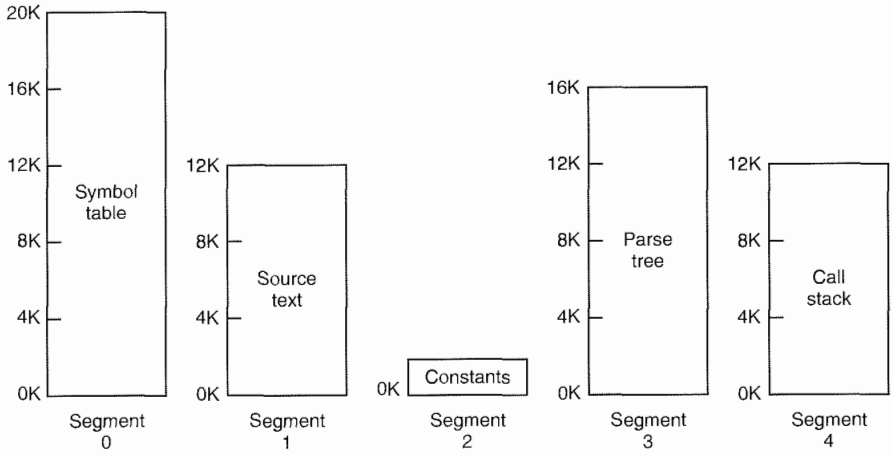


Figure 4-36. A segmented memory allows each table to grow or shrink independently of the other tables.

We emphasize that a segment is a logical entity, which the programmer is aware of and uses as a logical entity. A segment might contain a procedure, or an array, or a stack, or a collection of scalar variables, but usually it does not contain a mixture of different types.

A segmented memory has other advantages besides simplifying the handling of data structures that are growing or shrinking. If each procedure occupies a separate segment, with address 0 as its starting address, the linking up of procedures compiled separately is greatly simplified. After all the procedures that constitute a program have been compiled and linked up, a procedure call to the procedure in segment n will use the two-part address (n, 0) to address word 0 (the entry point).

If the procedure in segment n is subsequently modified and recompiled, no other procedures need be changed (because no starting addresses have been modified), even if the new version is larger than the old one. With a one-dimensional memory, the procedures are packed tightly next to each other, with no address space between them. Consequently, changing one procedure's size can affect the starting address of other, unrelated procedures. This, in turn, requires modifying all procedures that call any of the moved procedures, in order to incorporate their new starting addresses. If a program contains hundreds of procedures, this process can be costly.

Segmentation also facilitates sharing procedures or data between several processes. A common example is the **shared library**. Modern workstations that

run advanced window systems often have extremely large graphical libraries compiled into nearly every program. In a segmented system, the graphical library can be put in a segment and shared by multiple processes, eliminating the need for having it in every process' address space. While it is also possible to have shared libraries in pure paging systems, it is much more complicated. In effect, these systems do it by simulating segmentation.

Because each segment forms a logical entity of which the programmer is aware, such as a procedure, or an array, or a stack, different segments can have different kinds of protection. A procedure segment can be specified as execute only, prohibiting attempts to read from it or store into it. A floating-point array can be specified as read/write but not execute, and attempts to jump to it will be caught. Such protection is helpful in catching programming errors.

You should try to understand why protection makes sense in a segmented memory but not in a one-dimensional paged memory. In a segmented memory the user is aware of what is in each segment. Normally, a segment would not contain a procedure and a stack, for example, but one or the other. Since each segment contains only one type of object, the segment can have the protection appropriate for that particular type. Paging and segmentation are compared in Fig. 4-37.

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Figure 4-37. Comparison of paging and segmentation.

The contents of a page are, in a sense, accidental. The programmer is unaware of the fact that paging is even occurring. Although putting a few bits in each entry of the page table to specify the access allowed would be possible, to utilize this feature the programmer would have to keep track of where in his address space the page boundaries were. That is precisely the sort of administration that paging was invented to eliminate. Because the user of a segmented memory has the illusion that all segments are in main memory all the time—that is, he can address them as though they were—he can protect each segment separately, without having to be concerned with the administration of overlaying them.

4.8.1 Implementation of Pure Segmentation

The implementation of segmentation differs from paging in an essential way: pages are fixed size and segments are not. Figure 4-38(a) shows an example of physical memory initially containing five segments. Now consider what happens if segment 1 is evicted and segment 7, which is smaller, is put in its place. We arrive at the memory configuration of Fig. 4-38(b). Between segment 7 and segment 2 is an unused area—that is, a hole. Then segment 4 is replaced by segment 5, as in Fig. 4-38(c), and segment 3 is replaced by segment 6, as in Fig. 4-38(d). After the system has been running for a while, memory will be divided up into a number of chunks, some containing segments and some containing holes. This phenomenon, called **checkerboarding** or **external fragmentation**, wastes memory in the holes. It can be dealt with by compaction, as shown in Fig. 4-38(e).

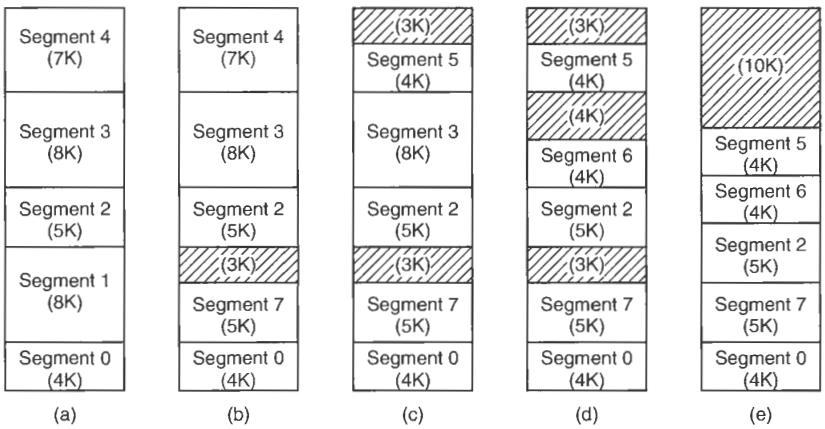


Figure 4-38. (a)-(d) Development of checkerboarding. (e) Removal of the checkerboarding by compaction.

4.8.2 Segmentation with Paging: MULTICS

If the segments are large, it may be inconvenient, or even impossible, to keep them in main memory in their entirety. This leads to the idea of paging them, so that only those pages that are actually needed have to be around. Several significant systems have supported paged segments. In this section we will describe the first one: MULTICS. In the next one we will discuss a more recent one: the Intel Pentium.

MULTICS ran on the Honeywell 6000 machines and their descendants and provided each program with a virtual memory of up to 2^{18} segments (more than 250,000), each of which could be up to 65,536 (36-bit) words long. To implement this, the MULTICS designers chose to treat each segment as a virtual memory and to page it, combining the advantages of paging (uniform page size and not having to keep the whole segment in memory if only part of it is being used) with the advantages of segmentation (ease of programming, modularity, protection, and sharing).

Each MULTICS program has a segment table, with one descriptor per segment. Since there are potentially more than a quarter of a million entries in the table, the segment table is itself a segment and is paged. A segment descriptor contains an indication of whether the segment is in main memory or not. If any part of the segment is in memory, the segment is considered to be in memory, and its page table will be in memory. If the segment is in memory, its descriptor contains an 18-bit pointer to its page table [see Fig. 4-39(a)]. Because physical addresses are 24 bits and pages are aligned on 64-byte boundaries (implying that the low-order 6 bits of page addresses are 000000), only 18 bits are needed in the descriptor to store a page table address. The descriptor also contains the segment size, the protection bits, and a few other items. Figure 4-39(b) illustrates a MULTICS segment descriptor. The address of the segment in secondary memory is not in the segment descriptor but in another table used by the segment fault handler.

Each segment is an ordinary virtual address space and is paged in the same way as the nonsegmented paged memory described earlier in this chapter. The normal page size is 1024 words (although a few small segments used by MULTICS itself are not paged or are paged in units of 64 words to save physical memory).

An address in MULTICS consists of two parts: the segment and the address within the segment. The address within the segment is further divided into a page number and a word within the page, as shown in Fig. 4-40. When a memory reference occurs, the following algorithm is carried out.

1. The segment number is used to find the segment descriptor.
2. A check is made to see if the segment's page table is in memory. If the page table is in memory, it is located. If it is not, a segment fault occurs. If there is a protection violation, a fault (trap) occurs.

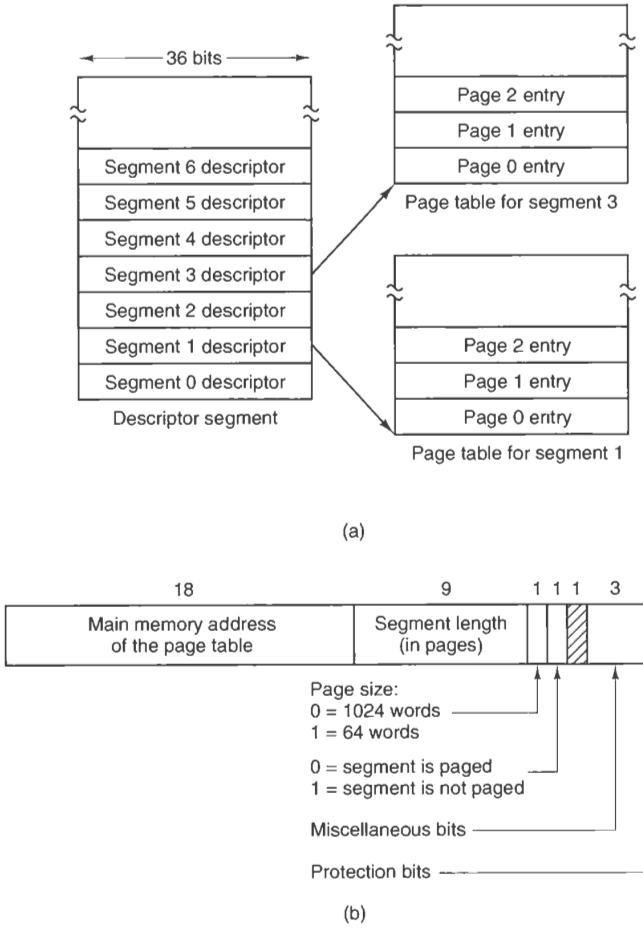


Figure 4-39. The MULTICS virtual memory. (a) The descriptor segment points to the page tables. (b) A segment descriptor. The numbers are the field lengths.

3. The page table entry for the requested virtual page is examined. If the page is not in memory, a page fault occurs. If it is in memory, the main memory address of the start of the page is extracted from the page table entry.
4. The offset is added to the page origin to give the main memory address where the word is located.
5. The read or store finally takes place.

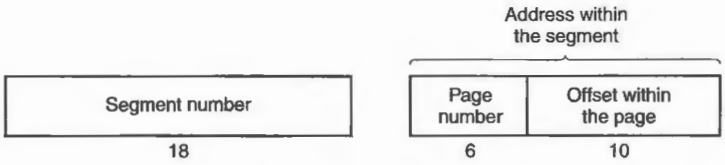


Figure 4-40. A 34-bit MULTICS virtual address.

This process is illustrated in Fig. 4-41. For simplicity, the fact that the descriptor segment is itself paged has been omitted. What really happens is that a register (the descriptor base register), is used to locate the descriptor segment's page table, which, in turn, points to the pages of the descriptor segment. Once the descriptor for the needed segment has been found, the addressing proceeds as shown in Fig. 4-41.

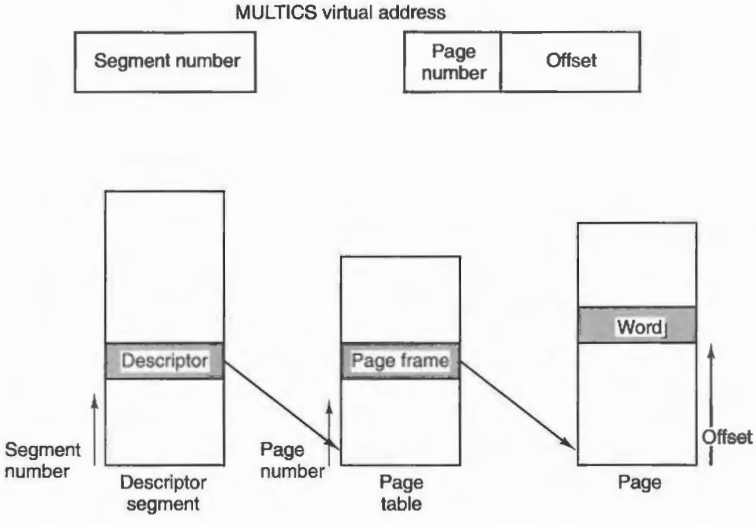


Figure 4-41. Conversion of a two-part MULTICS address into a main memory address.

As you have no doubt guessed by now, if the preceding algorithm were actually carried out by the operating system on every instruction, programs would not run very fast. In reality, the MULTICS hardware contains a 16-word high-speed TLB that can search all its entries in parallel for a given key. It is illustrated in Fig. 4-42. When an address is presented to the computer, the addressing hardware first checks to see if the virtual address is in the TLB. If so, it gets the page frame number directly from the TLB and forms the actual address of the referenced word without having to look in the descriptor segment or page table.

Comparison field					Is this entry used?
Segment number	Virtual page	Page frame	Protection	Age	
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Figure 4-42. A simplified version of the MULTICS TLB. The existence of two page sizes makes the actual TLB more complicated.

The addresses of the 16 most recently referenced pages are kept in the TLB. Programs whose working set is smaller than the TLB size will come to equilibrium with the addresses of the entire working set in the TLB and therefore will run efficiently. If the page is not in the TLB, the descriptor and page tables are actually referenced to find the page frame address, and the TLB is updated to include this page, the least recently used page being thrown out. The age field keeps track of which entry is the least recently used. The reason that a TLB is used is for comparing the segment and page number of all the entries in parallel.

4.8.3 Segmentation with Paging: The Intel Pentium

In many ways, the virtual memory on the Pentium resembles MULTICS, including the presence of both segmentation and paging. Whereas MULTICS has 256K independent segments, each up to 64K 36-bit words, the Pentium has 16K independent segments, each holding up to 1 billion 32-bit words. Although there are fewer segments, the larger segment size is far more important, as few programs need more than 1000 segments, but many programs need large segments.

The heart of the Pentium virtual memory consists of two tables, the **LDT** (**Local Descriptor Table**) and the **GDT** (**Global Descriptor Table**). Each program has its own LDT, but there is a single GDT, shared by all the programs on the computer. The LDT describes segments local to each program, including its code, data, stack, and so on, whereas the GDT describes system segments, including the operating system itself.

To access a segment, a Pentium program first loads a selector for that segment into one of the machine's six segment registers. During execution, the CS register holds the selector for the code segment and the DS register holds the selector for

the data segment. The other segment registers are less important. Each selector is a 16-bit number, as shown in Fig. 4-43.

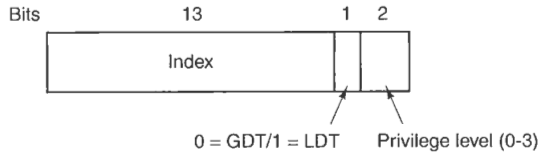


Figure 4-43. A Pentium selector.

One of the selector bits tells whether the segment is local or global (i.e., whether it is in the LDT or GDT). Thirteen other bits specify the LDT or GDT entry number, so these tables are each restricted to holding 8K segment descriptors. The other 2 bits relate to protection, and will be described later. Descriptor 0 is forbidden. It may be safely loaded into a segment register to indicate that the segment register is not currently available. It causes a trap if used.

At the time a selector is loaded into a segment register, the corresponding descriptor is fetched from the LDT or GDT and stored in microprogram registers, so it can be accessed quickly. A descriptor consists of 8 bytes, including the segment's base address, size, and other information, as depicted in Fig. 4-44.

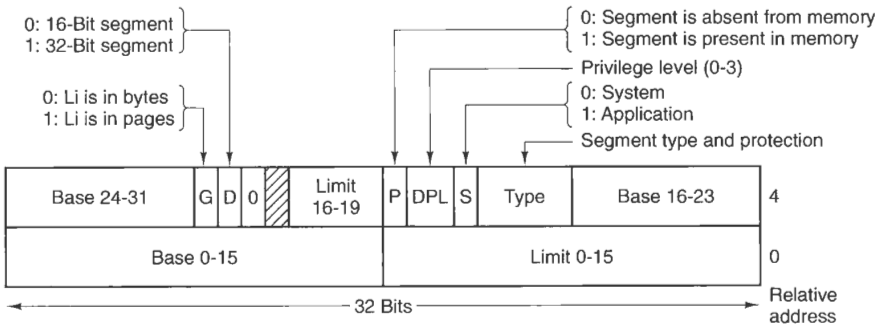


Figure 4-44. Pentium code segment descriptor. Data segments differ slightly.

The format of the selector has been cleverly chosen to make locating the descriptor easy. First either the LDT or GDT is selected, based on selector bit 2. Then the selector is copied to an internal scratch register, and the 3 low-order bits set to 0. Finally, the address of either the LDT or GDT table is added to it, to give a direct pointer to the descriptor. For example, selector 72 refers to entry 9 in the GDT, which is located at address GDT + 72.

Let us trace the steps by which a (selector, offset) pair is converted to a physical address. As soon as the microprogram knows which segment register is being

used, it can find the complete descriptor corresponding to that selector in its internal registers. If the segment does not exist (selector 0), or is currently paged out, a trap occurs.

It then checks to see if the offset is beyond the end of the segment, in which case a trap also occurs. Logically, there should simply be a 32-bit field in the descriptor giving the size of the segment, but there are only 20 bits available, so a different scheme is used. If the *Gbit* (Granularity) field is 0, the *Limit* field is the exact segment size, up to 1 MB. If it is 1, the *Limit* field gives the segment size in pages instead of bytes. The Pentium page size is fixed at 4 KB, so 20 bits are enough for segments up to 2^{32} bytes.

Assuming that the segment is in memory and the offset is in range, the Pentium then adds the 32-bit *Base* field in the descriptor to the offset to form what is called a **linear address**, as shown in Fig. 4-45. The *Base* field is broken up into three pieces and spread all over the descriptor for compatibility with the 286, in which the *Base* is only 24 bits. In effect, the *Base* field allows each segment to start at an arbitrary place within the 32-bit linear address space.

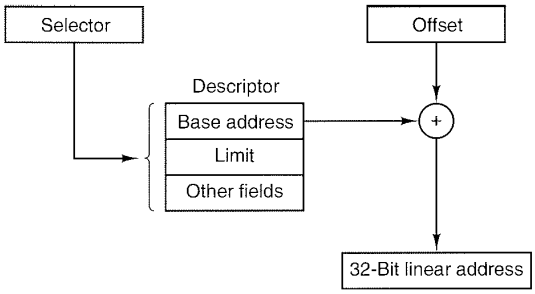


Figure 4-45. Conversion of a (selector, offset) pair to a linear address.

If paging is disabled (by a bit in a global control register), the linear address is interpreted as the physical address and sent to the memory for the read or write. Thus with paging disabled, we have a pure segmentation scheme, with each segment's base address given in its descriptor. Segments are permitted to overlap, incidentally, probably because it would be too much trouble and take too much time to verify that they were all disjoint.

On the other hand, if paging is enabled, the linear address is interpreted as a virtual address and mapped onto the physical address using page tables, pretty much as in our earlier examples. The only real complication is that with a 32-bit virtual address and a 4-KB page, a segment might contain 1 million pages, so a two-level mapping is used to reduce the page table size for small segments.

Each running program has a **page directory** consisting of 1024 32-bit entries. It is located at an address pointed to by a global register. Each entry in this

directory points to a page table also containing 1024 32-bit entries. The page table entries point to page frames. The scheme is shown in Fig. 4-46.

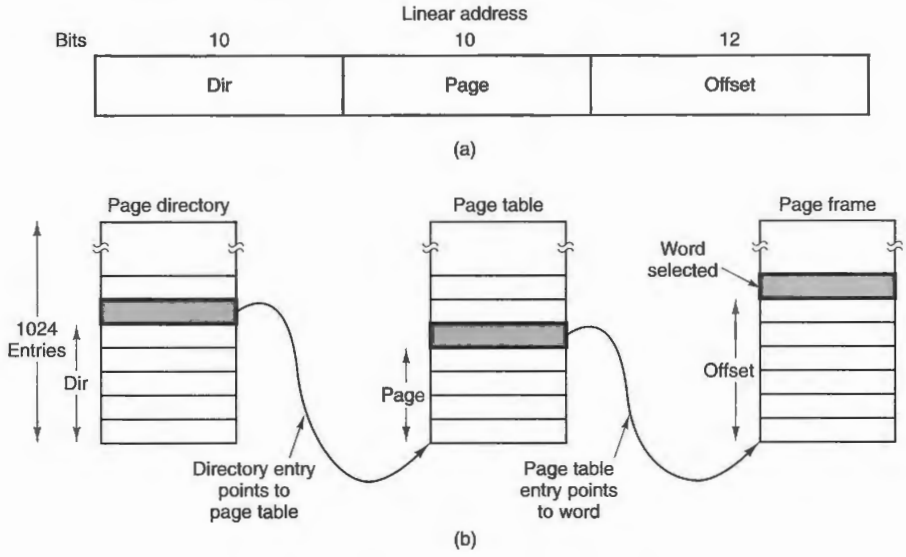


Figure 4-46. Mapping of a linear address onto a physical address.

In Fig. 4-46(a) we see a linear address divided into three fields, *Dir*, *Page*, and *Offset*. The *Dir* field is used to index into the page directory to locate a pointer to the proper page table. Then the *Page* field is used as an index into the page table to find the physical address of the page frame. Finally, *Offset* is added to the address of the page frame to get the physical address of the byte or word needed.

The page table entries are 32 bits each, 20 of which contain a page frame number. The remaining bits contain access and dirty bits, set by the hardware for the benefit of the operating system, protection bits, and other utility bits.

Each page table has entries for 1024 4-KB page frames, so a single page table handles 4 megabytes of memory. A segment shorter than 4M will have a page directory with a single entry, a pointer to its one and only page table. In this way, the overhead for short segments is only two pages, instead of the million pages that would be needed in a one-level page table.

To avoid making repeated references to memory, the Pentium, like MULTICS, has a small TLB that directly maps the most recently used *Dir–Page* combinations onto the physical address of the page frame. Only when the current combination is not present in the TLB is the mechanism of Fig. 4-46 actually carried out and the TLB updated. As long as TLB misses are rare, performance is good.

It is also worth noting that if some application does not need segmentation but is content with a single, paged, 32-bit address space, that model is possible. All the segment registers can be set up with the same selector, whose descriptor has *Base* = 0 and *Limit* set to the maximum. The instruction offset will then be the linear address, with only a single address space used—in effect, normal paging. In fact, all current operating systems for the Pentium work this way. OS/2 was the only one that used the full power of the Intel MMU architecture.

All in all, one has to give credit to the Pentium designers. Given the conflicting goals of implementing pure paging, pure segmentation, and paged segments, while at the same time being compatible with the 286, and doing all of this efficiently, the resulting design is surprisingly simple and clean.

Although we have covered the complete architecture of the Pentium virtual memory, albeit briefly, it is worth saying a few words about protection, since this subject is intimately related to the virtual memory. Just as the virtual memory scheme is closely modeled on MULTICS, so is the protection system. The Pentium supports four protection levels with level 0 being the most privileged and level 3 the least. These are shown in Fig. 4-47. At each instant, a running program is at a certain level, indicated by a 2-bit field in its PSW. Each segment in the system also has a level.

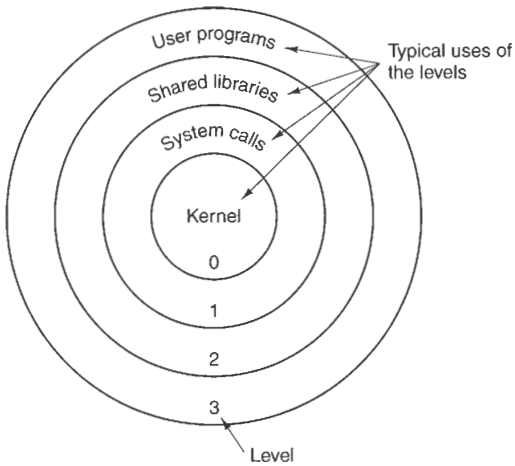


Figure 4-47. Protection on the Pentium.

As long as a program restricts itself to using segments at its own level, everything works fine. Attempts to access data at a higher level are permitted. Attempts to access data at a lower level are illegal and cause traps. Attempts to call procedures at a different level (higher or lower) are allowed, but in a carefully controlled way. To make an interlevel call, the CALL instruction must contain a selector instead of an address. This selector designates a descriptor called a **call**

gate, which gives the address of the procedure to be called. Thus it is not possible to jump into the middle of an arbitrary code segment at a different level. Only official entry points may be used. The concepts of protection levels and call gates were pioneered in MULTICS, where they were viewed as **protection rings**.

A typical use for this mechanism is suggested in Fig. 4-47. At level 0, we find the kernel of the operating system, which handles I/O, memory management, and other critical matters. At level 1, the system call handler is present. User programs may call procedures here to have system calls carried out, but only a specific and protected list of procedures may be called. Level 2 contains library procedures, possibly shared among many running programs. User programs may call these procedures and read their data, but they may not modify them. Finally, user programs run at level 3, which has the least protection.

Traps and interrupts use a mechanism similar to the call gates. They, too, reference descriptors, rather than absolute addresses, and these descriptors point to specific procedures to be executed. The *Type* field in Fig. 4-44 distinguishes between code segments, data segments, and the various kinds of gates.

4.9 RESEARCH ON MEMORY MANAGEMENT

Memory management, especially paging algorithms, was once a fruitful area for research, but most of that seems to have largely died off, at least for general-purpose systems. Most real systems tend to use some variation on clock, because it is easy to implement and relatively effective. One recent exception, however, is a redesign of the 4.4 BSD virtual memory system (Cranor and Parulkar, 1999).

There is still research going on concerning paging in special-purpose and newer kinds of systems though. Some of this work relates to allowing user processes to field their own page faults and handle their own memory management, possibly in an application-specific way (Engler et al., 1995). One area where applications may need to do their own paging in special ways is multimedia, so some research has looked at this (Hand, 1999). Another area that has some special requirements is hand-held personal communicators (Abutaleb and Li, 1997; and Wan and Lin, 1997). One final area is systems with 64-bit address spaces shared by many processes (Talluri et al., 1995).

4.10 SUMMARY

In this chapter we have examined memory management. We saw that the simplest systems do not swap or page at all. Once a program is loaded into memory, it remains there until it finishes. Some operating systems allow only one process at a time in memory, while others support multiprogramming.

The next step up is swapping. When swapping is used, the system can handle more processes than it has room for in memory. Processes for which there is no room are swapped out to the disk. Free space in memory and on disk can be kept track of with a bitmap or a hole list.

Modern computers often have some form of virtual memory. In the simplest form, each process' address space is divided up into uniform sized blocks called pages, which can be placed into any available page frame in memory. There are many page replacement algorithms; two of the better algorithms are aging and WSClock.

Paging systems can be modeled by abstracting the page reference string from the program and using the same reference string with different algorithms. These models can be used to make some predictions about paging behavior.

To make paging systems work well, choosing an algorithm is not enough; attention to issues such as determining the working set, memory allocation policy, and page size are required.

Segmentation helps in handling data structures that change size during execution and simplifies linking and sharing. It also facilitates providing different protection for different segments. Sometimes segmentation and paging are combined to provide a two-dimensional virtual memory. The MULTICS system and the Intel Pentium support segmentation and paging.

PROBLEMS

1. A computer system has enough room to hold four programs in its main memory. These programs are idle waiting for I/O half the time. What fraction of the CPU time is wasted?

2. In Fig. 4-21 we saw an example of how multiple jobs can run in parallel and finish faster than if they had run sequentially. Suppose that two jobs, each of which needs 10 minutes of CPU time, start simultaneously. How long will the last one take to complete if they run sequentially? How long if they run in parallel? Assume 50% I/O wait.

3. A swapping system eliminates holes by compaction. Assuming a random distribution of many holes and many data segments and a time to read or write a 32-bit memory word of 10 nsec, about how long does it take to compact 128 MB? For simplicity, assume that word 0 is part of a hole and that the highest word in memory contains valid data.

4. In this problem you are to compare the storage needed to keep track of free memory using a bitmap versus using a linked list. The 128-MB memory is allocated in units of

n bytes. For the linked list, assume that memory consists of an alternating sequence of segments and holes, each 64 KB. Also assume that each node in the linked list needs a 32-bit memory address, a 16-bit length, and a 16-bit next-node field. How many bytes of storage is required for each method? Which one is better?

5. Consider a swapping system in which memory consists of the following hole sizes in memory order: 10 KB, 4 KB, 20 KB, 18 KB, 7 KB, 9 KB, 12 KB, and 15 KB. Which hole is taken for successive segment requests of

(a) 12 KB
(b) 10 KB
(c) 9 KB

for first fit? Now repeat the question for best fit, worst fit, and next fit.

6. What is the difference between a physical address and a virtual address?

7. For each of the following decimal virtual addresses, compute the virtual page number and offset for a 4-KB page and for an 8 KB page: 20000, 32768, 60000.

8. Using the page table of Fig. 4-10, give the physical address corresponding to each of the following virtual addresses:

(a) 20
(b) 4100
(c) 8300

9. The Intel 8086 processor does not support virtual memory. Nevertheless, some companies previously sold systems that contained an unmodified 8086 CPU and do paging. Make an educated guess as to how they did it. *Hint*: Think about the logical location of the MMU.

10. The amount of disk space that must be available for page storage is related to the maximum number of processes, n, the number of bytes in the virtual address space, v, and the number of bytes of RAM, r. Give an expression for the worst case disk space requirements. How realistic is this amount?

11. If an instruction takes 10 nsec and a page fault takes an additional n nsec, give a formula for the effective instruction time if page faults occur every k instructions.

12. A machine has a 32-bit address space and an 8-KB page. The page table is entirely in hardware, with one 32-bit word per entry. When a process starts, the page table is copied to the hardware from memory, at one word every 100 nsec. If each process runs for 100 msec (including the time to load the page table), what fraction of the CPU time is devoted to loading the page tables?

13. A computer with a 32-bit address uses a two-level page table. Virtual addresses are split into a 9-bit top-level page table field, an 11-bit second-level page table field, and an offset. How large are the pages and how many are there in the address space?

14. Suppose that a 32-bit virtual address is broken up into four fields, a, b, c, and d. The first three are used for a three-level page table system. The fourth field, d, is the offset. Does the number of pages depend on the sizes of all four fields? If not, which ones matter and which ones do not?

15. A computer has 32-bit virtual addresses and 4-KB pages. The program and data together fit in the lowest page (0–4095) The stack fits in the highest page. How many entries are needed in the page table if traditional (one-level) paging is used? How many page table entries are needed for two-level paging, with 10 bits in each part?

16. Below is an execution trace of a program fragment for a computer with 512-byte pages. The program is located at address 1020, and its stack pointer is at 8192 (the stack grows toward 0). Give the page reference string generated by this program. Each instruction occupies 4 bytes (1 word) including immediate constants. Both instruction and data references count in the reference string.

Load word 6144 into register 0
Push register 0 onto the stack
Call a procedure at 5120, stacking the return address
Subtract the immediate constant 16 from the stack pointer
Compare the actual parameter to the immediate constant 4
Jump if equal to 5152

17. A computer whose processes have 1024 pages in their address spaces keeps its page tables in memory. The overhead required for reading a word from the page table is 5 nsec. To reduce this overhead, the computer has a TLB, which holds 32 (virtual page, physical page frame) pairs, and can do a look up in 1 nsec. What hit rate is needed to reduce the mean overhead to 2 nsec?

18. The TLB on the VAX does not contain an *R* bit. Why?

19. How can the associative memory device needed for a TLB be implemented in hardware, and what are the implications of such a design for expandability?

20. A machine has 48-bit virtual addresses and 32-bit physical addresses. Pages are 8 KB. How many entries are needed for the page table?

21. A computer with an 8-KB page, a 256-KB main memory, and a 64-GB virtual address space uses an inverted page table to implement its virtual memory. How big should the hash table be to ensure a mean hash chain length of less than 1? Assume that the hash table size is a power of two.

22. A student in a compiler design course proposes to the professor a project of writing a compiler that will produce a list of page references that can be used to implement the optimal page replacement algorithm. Is this possible? Why or why not? Is there anything that could be done to improve paging efficiency at run time?

23. If FIFO page replacement is used with four page frames and eight pages, how many page faults will occur with the reference string 0172327103 if the four frames are initially empty? Now repeat this problem for LRU.

24. Consider the page sequence of Fig. 4-16(b). Suppose that the *R* bits for the pages *B* through *A* are 11011011, respectively. Which page will second chance remove?

25. A small computer has four page frames. At the first clock tick, the *R* bits are 0111 (page 0 is 0, the rest are 1). At subsequent clock ticks, the values are 1011, 1010, 1101, 0010, 1010, 1100, and 0001. If the aging algorithm is used with an 8-bit counter, give the values of the four counters after the last tick.

26. Suppose that $\tau = 400$ in Fig. 4-21. Which page will be removed?

27. In the WSClock algorithm of Fig. 4-22(c), the hand points to a page with $R = 0$. If $\tau = 400$, will this page be removed? What about if $\tau = 1000$?

28. How long does it take to load a 64-KB program from a disk whose average seek time is 10 msec, whose rotation time is 10 msec, and whose tracks hold 32 KB

(a) for a 2-KB page size?
(b) for a 4-KB page size?

The pages are spread randomly around the disk and the number of cylinders is so large that the chance of two pages being on the same cylinder is negligible.

29. A computer has four page frames. The time of loading, time of last access, and the *R* and *M* bits for each page are as shown below (the times are in clock ticks):

Page	Loaded	Last ref.	R	M
0	126	280	1	0
1	230	265	0	01
2	140	270	0	0
3	110	285	1	1

(a) Which page will NRU replace?
(b) Which page will FIFO replace?
(c) Which page will LRU replace?
(d) Which page will second chance replace?

30. One of the first timesharing machines, the PDP-1, had a memory of 4K 18-bit words. It held one process at a time in memory. When the scheduler decided to run another process, the process in memory was written to a paging drum, with 4K 18-bit words around the circumference of the drum. The drum could start writing (or reading) at any word, rather than only at word 0. Why do you suppose this drum was chosen?

31. A computer provides each process with 65,536 bytes of address space divided into pages of 4096 bytes. A particular program has a text size of 32,768 bytes, a data size of 16,386 bytes, and a stack size of 15,870 bytes. Will this program fit in the address space? If the page size were 512 bytes, would it fit? Remember that a page may not contain parts of two different segments.

32. Can a page be in two working sets at the same time? Explain.

33. If a page is shared between two processes, is it possible that the page is read-only for one process and read-write for the other? Why or why not?

34. It has been observed that the number of instructions executed between page faults is directly proportional to the number of page frames allocated to a program. If the available memory is doubled, the mean interval between page faults is also doubled. Suppose that a normal instruction takes 1 microsec, but if a page fault occurs, it takes 2001 μsec (i.e., 2 msec to handle the fault). If a program takes 60 sec to run, during which time it gets 15,000 page faults, how long would it take to run if twice as much memory were available?

35. A group of operating system designers for the Frugal Computer Company are thinking about ways of reducing the amount of backing store needed in their new operating system. The head guru has just suggested not bothering to save the program text in the swap area at all, but just page it in directly from the binary file whenever it is needed. Under what conditions, if any, does this idea work for the program text? Under what conditions, if any, does it work for the data?

36. A machine language instruction to load a 32-bit word into a register contains the 32-bit address of the word to be loaded. What is the maximum number of page faults this instruction can cause?

37. Explain the difference between internal fragmentation and external fragmentation. Which one occurs in paging systems? Which one occurs in systems using pure segmentation?

38. When segmentation and paging are both being used, as in MULTICS, first the segment descriptor must be looked up, then the page descriptor. Does the TLB also work this way, with two levels of lookup?

39. Plot a histogram and calculate the mean and median of the sizes of executable binary files on a computer to which you have access. On a Windows system, look at all .exe and .dll files; on a UNIX system look at all executable files in */bin*, */usr/bin*, and */local/bin* that are not scripts (or use the *file* utility to find all executables). Determine the optimal page size for this computer just considering the code (not data). Consider internal fragmentation and page table size, making some reasonable assumption about the size of a page table entry. Assume that all programs are equally likely to be run and thus should be weighted equally.

40. Small programs for MS-DOS can be compiled as *.COM* files. These files are always loaded at address 0x100 in a single memory segment that is used for code, data, and stack. Instructions that transfer control of execution, such as JMP and CALL, or that access static data from fixed addresses have the addresses compiled into the object code. Write a program that can relocate such a program file to run starting at an arbitrary address. Your program must scan through code looking for object codes for instructions that refer to fixed memory addresses, then modify those addresses that point to memory locations within the range to be relocated. You can find the object codes in an assembly language programming text. Note that doing this perfectly without additional information is, in general, an impossible task, because some data words may have values that mimic instruction object codes.

41. Write a program that simulates a paging system. At the start of the program, the user should be asked to choose a page replacement algorithm, choosing from FIFO, LRU, and at least one other. On each cycle, read the number of the referenced page from a file. Generate a listing similar to Fig. 4-25, except rotated 90 degrees so that each new page reference increases the length of the output file by one line.

42. Write a program that models the distance string algorithm described in the text. The input to the program is a list of page references (contained in a file), plus the number of page frames of physical memory available. If possible, use trace data from real programs instead of randomly-generated page references. The program should maintain the stack of pages, analogous to Fig. 4-25. At each page fault, a procedure should

be called to choose a replacement page. When the run is completed, the program should plot the distance string, analogous to Fig. 4-26. Make multiple runs for different values of the memory size and see what conclusions you can draw.